CHIP WISSEN

Hans-Joachim Sacht

µP-Programmierfibel für 2650/6502/6800/8080-85

Einführung in die Programmiertechnik

Ein Buch von CHIP, dem Mikrocomputer-Fachmagazin

Hans-Joachim Sacht
µP-Programmierfibel
für

2650

6502

6800

8080-85

CHIP WISSEN

Dr. Hans-Joachim Sacht

µP-Programmierfibel für 2650/6502/6800/8080-85

Einführung in die Programmiertechnik

2., durchgesehene und korrigierte Auflage

VOGEL-BUCHVERLAG
WÜRZBURG

Dr. HANS-JOACHIM SACHT
hat in Berlin Maschinenbau und in Hamburg Beriebswirtschaftslehre studiert. Schon im Krieg hat er Berührung mit der „Elektronik" bekommen, als er eins der ersten deutschen Radargeräte im Einsatz führte. Nach einigen Jahren in der Schweiz und in Amerika war er in leitender Position in der deutschen Industrie, zuletzt langjährig als Geschäftsführer einer Waagenfabrik, tätig. Dort hat er den Übergang von der Mechanik zur Elektronik erlebt und mitbeeinflußt. Heute arbeitet er als Unternehmensberater auf dem Gebiet der praktischen Mikroprozessoranwendung und hat in zahlreichen Seminaren und Veröffentlichungen auch viele Nichtfachleute in dieses Gebiet eingeführt.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Sacht, Hans-Joachim:
µP-Programmierfibel [MyP-Programmierfibel] für 2650/6502/6800/8080-85: Einf. in d. Programmiertechnik/Hans-Joachim Sacht. – 2., durchges. u. korrigierte Aufl. – Würzburg: Vogel, 1984.
(Chip-Wissen)
ISBN 3-8023-0644-9

ISBN 3-8023-0644-9
2. Auflage. 1984
Alle Rechte, auch der Übersetzung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verarbeitet werden.
Printed in Germany
Copyright 1980 by Vogel-Buchverlag, Würzburg
Herstellung: Vogel-Druck Würzburg

Inhaltsverzeichnis

Vorwort 11
Einführung in den Inhalt des Buches 13

1 Grundlagen für die Programmierarbeit 19
1.1 Die Bauelemente eines Mikrocomputers 19
1.2 Verschiedene Systeme zur Darstellung von Zahlen 24
1.3 Das Programm als Folge von Befehlen 29
1.4 Befehlsablauf und Speicherorganisation 31
1.5 Takt, Maschinenzyklus und Befehlsausführungszeit 35
1.6 Struktureller Aufbau eines Mikroprozessors 38

2 Etwas Binär-Mathematik 41
2.1 Addition und Subtraktion ganzer Zahlen 42
2.2 Das Zweier-Komplementsystem 45
2.3 Multiplikation und Division 56
2.4 Addition und Subtraktion im BCD-Zahlensystem 60
2.5 Rechnen mit Dezimalstellen 64
2.6 Umwandlung der verschiedenen Zahlensysteme 66

3 Aufbau und Wirkung der Befehle für Mikroprozessoren .. 69
3.1 Maschinensprache, Assemblersprache 70
3.2 Befehlsbeschreibung und Befehlssatz 73
3.3 Befehlsformat und Befehlsaufbau 75

3.4 Die vier wichtigsten Befehlsarten 77
3.4.1 Datentransport-Befehle 77
3.4.2 Datenverarbeitungs-Befehle 79
3.4.3 Prüf- und Vergleichs-Befehle 81
3.4.4 Programmablauf-Befehle 82
3.5 Die Bedeutung des Programm-Status-Worts 85
3.6 Möglichkeiten der Datenverarbeitung 87
3.6.1 Addition und Subtraktion im Zweier-Komplement 87
3.6.2 Logische Verknüpfungen 90
3.6.3 Rotieren und Schieben 92
3.7 Unterprogrammanruf und Stapelspeicherbedienung 93
3.8 Die verschiedenen Adressierarten 96
3.8.1 Registeradressierung 97
3.8.2 Adressierung durch Registerpaar 98
3.8.3 Unmittelbare Adressierung 99
3.8.4 Absolute Adressierung 99
3.8.5 Seite-Null-Adressierung 101
3.8.6 Relative Adressierung 102
3.8.7 Indirekte Adressierung 104
3.8.8 Index-Adressierung 105
3.9 Ein- und Ausgabe-Operationen 107
3.10 Behandlung von Programmunterbrechungen (Interrupt) .. 110

4 Die Befehlssätze der Mikroprozessoren 2650, 6502, 6800 und 8080/85 113
4.1 Befehlstabellen und dort verwendete Symbole 114
4.2 Durch Befehle zugängliche Register 115
4.3 Datentransport-Befehle 118
4.3.1 Ein-Byte-Befehle mit Register-Adressierung 118
4.3.2 Zwei-Byte-Befehle mit unmittelbarer Adressierung 121
4.3.3 Drei-Byte-Befehle mit absoluter Adressierung 123
4.3.4 Zwei-Byte-Befehle mit Seite-Null-Adressierung 129
4.3.5 Zwei-Byte-Befehle mir relativer Adressierung 130
4.3.6 Befehle mit indirekter Adressierung 131
4.3.7 Befehle mit Index-Adressierung 133
4.3.8 Kombination von indirekter und Index-Adressierung 137
4.4 Datenverarbeitungs-Befehle 139
4.4.1 Befehle für die Verknüpfung von zwei Operanden 140

4.4.2 Increment-, Decrement- und Löschbefehle 146
4.4.3 Rotations- und Schiebebefehle 149
4.5 Ein- und Ausgabebefehle 152
4.6 Vergleichs- und Testbefehle 154
4.7 Status-Register-Befehle 159
4.8 Befehle zur Bedienung des Stapelspeichers 163
4.9 Programmablaufbefehle 165
4.9.1 Absolute Sprungbefehle 165
4.9.2 Bedingte Sprungbefehle 167
4.9.3 Befehle für den Anruf von Unterprogrammen 169
4.10 Befehle zur Dezimalkorrektur 173
4.11 Sonstige Befehle 175
4.12 Einrichtungen für den Programmstart 176

5 Beispiele für Mikroprozessor-Programme (Lernprogramme) 179
5.1 Addition und Subtraktion von Binärzahlen
Initialisieren. Arbeit mit und ohne Carry. Laden und Speichern. Benutzung von Additions- und Subtraktions-Befehlen. Setzen eines Merkers. 184
5.2 Verschachtelte Schleifen zur Zeitverzögerung
Anruf von Unterprogrammen. Retten von Registerinhalten. Stapelspeicher-Bedienung. Parameter-Übergabe. Programmierung von Schleifen. Incrementieren und Decrementieren. Zeitberechnung. Symbolische Adressen. Berechnung von relativen Sprüngen. 191
5.3 Der Größe nach sortieren
JA/NEIN-Entscheidungen. Logische Vergleiche. Bedingte Sprünge. Anwendung von Vergleichsbefehlen. Aufbau eines Hilfsregisters. Bedingte und ewige Schleifen. 202
5.4 Ersatz einer Relaissteuerung durch Programm
Festlegen von Ein- und Ausgabe-Port. Baustein-Initialisierung. Anwenden von Testbefehlen. Prüfen von Teilen eines Bitmusters. Anwenden von Ein- und Ausgabebefehlen. 208
5.5 Codewandlung durch Rechnung
Aufbau eines Rechen-Algorithmus. Anwenden von Vergleichsbefehlen. Bit-Maskierung. Logisches UND. Logisches ODER. Festlegen von Sprunggrenzen. 217

5.6 Speicherbereich auf Null setzen
Verschiedene Möglichkeiten der Index-Adressierung. Berechnung einer Anzahl von Speicherplätzen. Berechnung von effektiven Adressen bei Indexbefehlen. 223
5.7 Datenfeld (Block) im Speicher durchsuchen
Definition eines Datenfeldes. Einrichtung einer Suchschleife. Aufbau eines Adreß-Zeiger-Registers. 232
5.8 Bedingter Einsprung in Unterprogramme
Anwenden von Rotationsbefehlen. Indirekte Adressierung. Tabelle mit Einsprung-Adreßzeigern. Einsprung- und Rücksprungbefehle für Unterprogramme. 237
5.9 16-Bit-Addition von zwei Datenfeldern
Verarbeiten von Datenfeldern. Mehr-Byte-Rechnung. Berücksichtigung des Übertrags. Mehrfach-Adressierung mit Index. 247
5.10 BCD-Addition und -Subtraktion
Befehle für Dezimalkorrektur. Unterschiedliche Behandlung des Carry-Bit bei der Subtraktion. Prüfen, ob Übertrag oder Borgen entstanden ist. 252

6 Struktur und Befehlsaufbau einiger weiterer Mikroprozessor-Typen 261
6.1 Der Ein-Chip-Computer 8048 262
6.2 Der 8-Bit-Mikroprozessor Z 80 268
6.3 Der 16-Bit-Mikroprozessor 9900 271

7 Hinweise für die praktische Programmierarbeit 279
7.1 Programmieren lernen 281
7.2 Anforderungen an Mikroprozessor-Lern- und Übungsgeräte 284
7.2.1 Hardware-Aufbau 284
7.2.2 Anforderungen an das Monitorprogramm 285
7.3 Die Schritte der Programmentwicklung 287
7.3.1 Aufgabenbeschreibung 289
7.3.2 Festlegung der Hardware 290
7.3.3 Beschreibendes Flußdiagramm 291
7.3.4 Befehlsdiagramm und Programmentwurf 292
7.3.5 Codieren, Editieren und Assemblieren 295
7.3.6 Prüfen, Testen und Fehlerbeseitigen 299

7.3.7 Anfertigung der Dokumentation 301
7.4 Strukturierte Programmierung 302
7.5 Programmanalyse 306

8 Beispiel für das Vorgehen bei einer Programmentwicklung 311
8.1 Aufgabenstellung 311
8.2 Hardwarekonzept 312
8.3 Problemlösung suchen 313
8.4 Programmentwicklung 318
8.5 Prüfen der Programmodule 325
8.6 Endprüfung und Dokumentation 326

Anhang 329

ASCII-Code in hexadezimaler Darstellung 329
Vergleichstabelle: Hexadezimal – Dezimal 329
Tabelle zur Ermittlung relativer Verschiebungen 331
Darstellung von positiven und negativen Dezimalzahlen hexadezimal codiert 331
Befehlstabellen 2650 332
Befehlstabellen 6502 336
Befehlstabellen 6800 340
Befehlstabellen 8080 344
Mikroprozessor-Lern- und Übungsgeräte 348
Bezugsquellenverzeichnis für Lern- und Übungsgeräte 352

Stichwortverzeichnis 353

Vorwort

Für jeden, der sich mit der Mikroprozessor-Technik zum erstenmal beschäftigt, ist es kaum erklärlich, wie ein Mikrocomputer mit den wenigen einfachen Befehlen, die er ausführen kann, in der Lage ist, auch sehr komplizierte Rechen- und Steuerungsaufgaben zu erfüllen.

Aufgabe dieses Buches soll es sein, Anfängern, die noch nichts oder nur wenig vom Programmieren wissen, die Einarbeitung in die Programmiertechnik für Mikroprozessoren zu erleichtern. Es werden darum insbesondere die Grundbegriffe, die man für das Verstehen des Programmaufbaus braucht, die aber in Fachbüchern und Firmenschriften oft vorausgesetzt werden, erklärt.

Da jeder, der sich mit der Technik oder den Naturwissenschaften beschäftigt, in Zukunft auch mit dem Mikroprozessor zu tun haben wird, soll dieses Buch den Studierenden aller technischen Fachrichtungen nützlich sein, aber auch Technikern und Ingenieuren dienen, die ihre Ausbildung schon abgeschlossen haben, als es noch keinen Mikroprozessor gab.

Das Buch will auch denjenigen in Management und Entwicklung helfen, die zwar nicht selbst programmieren wollen, die aber als Anwender Aufgaben für ein Mikroprozessor-System formulieren müssen, was sehr erleichtert wird, wenn Kenntnisse über die Programmiertechnik vorhanden sind.

Im Buch werden die Befehlssätze verschiedener Prozessoren erklärt und verglichen, wobei auch die gängigen Fachausdrücke der amerikanischen Computersprache erläutert werden. Es dürfte darum auch für alle,

die noch den richtigen Prozessor suchen oder die von einem Typ auf den anderen umsteigen wollen, viele Anregungen bringen.

An einer großen Anzahl von Beispielen wird gezeigt, wie Befehle anzuwenden sind, um gestellte Aufgaben zu lösen. Der praktische Aufbau von Programmen wird besprochen, und Programmiertricks werden erklärt. Das Entstehen eines Programms wird Schritt für Schritt dargestellt.

Für diejenigen, die nur wissen wollen, wie ein Mikroprozessor-System programmiert wird, dürfte es ausreichen, das Buch mit Bleistift und Papier durchzuarbeiten. Richtiges Programmieren kann man allerdings nur mit Hilfe eines arbeitenden Mikroprozessors lernen.

Hierfür sehr zu empfehlen ist die Arbeit mit einem Mikroprozessor-Lern- und Übungsgerät, von denen einige im Anhang des Buches vorgestellt werden.

Duisburg *Hans-Joachim Sacht*

Hinweise für den Leser

Einteilung des Inhalts

Aufgabe dieses Buches ist es, den Leser in die Programmiertechnik von Mikroprozessoren einzuführen. Es befaßt sich darum in erster Linie auch nur mit dem, was man jetzt allgemein als *Software* bezeichnet.

Für das Verständnis der Programmiertechnik sind jedoch gewisse Grundkenntnisse der *Hardware* über den Aufbau der Bauelemente eines Mikrocomputers und ihr Zusammenwirken ebenso notwendig, wie es Voraussetzung ist, daß die verschiedenen in der Mikroprozessor-Technik verwendeten *Zahlensysteme* bekannt sind. Im *Kapitel 1* dieses Buches werden darum diese Voraussetzungen für die Programmierarbeit in kurzen Zusammenfassungen behandelt.

Da alle Mikroprozessoren intern im *binären Zahlensystem* arbeiten, sollte ein Programmierer die hier geltenden Regeln für das Rechnen und Umcodieren kennen. Das *Kapitel 2* vermittelt dieses Wissen, soweit es für das Verständnis der später besprochenen Befehlssätze und Programmbeispiele notwendig ist.

Wirkung und Aufbau der Befehle sind bei den einzelnen Mikroprozessor-Typen unterschiedlich. Es gibt hierbei jedoch gewisse Grundregeln, die für alle Prozessoren gelten. Sie werden im *Kapitel 3* erklärt.

Wer ein Programm für einen Mikrocomputer schreiben will, muß den *Befehlssatz* des verwendeten Typs beherrschen. Dies bedeutet, er muß im einzelnen wissen, was geschieht, wenn ein bestimmter Befehl zur Verarbeitung kommt. Im *Kapitel 4* werden die Befehlssätze von vier in Deutschland häufig verwendeten 8-Bit-Mikroprozessoren erklärt und

miteinander verglichen, wobei alle Befehle sowohl in der Maschinen- als auch in der Assemblersprache dargestellt werden.

Ein Mikroprozessor-Programm setzt sich im allgemeinen aus vielen kleinen Programmblöcken zusammen, durch die jeweils immer eine ganz bestimmte Teilaufgabe gelöst wird. Im *Kapitel 5* wird an *10 Lernprogrammen* gezeigt, wie Einzelaufgaben, die immer wieder vorkommen, durch das Zusammenwirken geeigneter Befehle gelöst werden können. Jedes der Programmbeispiele wird, in vier verschiedenen Versionen programmiert, jeweils mit den Befehlen eines der vier besprochenen Mikroprozessoren vorgestellt. Dabei wird eine große Anzahl von Einzelheiten, die für die Programmierung wichtig sind, ausführlich besprochen, und es wird erklärt, wie die gleiche Aufgabe mit den verschiedenen Prozessoren gelöst werden kann.

In Kapitel 6 werden einige *weitere Mikroprozessor-Typen* behandelt, die erst in jüngster Zeit entwickelt wurden und die für besondere Aufgaben mit Vorteil Verwendung finden können.

Das Kapitel 7 gibt grundsätzliche Hinweise, wie man in der Praxis programmieren kann. Es werden die verschiedenen *Arbeitsmethoden für die Programmentwicklung* besprochen und gezeigt, wie man bei der praktischen Arbeit vorgehen kann.

In Kapitel 8 wird erklärt, wie eine in sich abgeschlossene Aufgabe durch ein Mikroprozessor-Programm gelöst wird. Hierbei werden alle notwendigen Überlegungen eingehend erörtert, und die *Entstehung des Programms Schritt für Schritt* mit allen dabei auftretenden Schwierigkeiten gezeigt.

Der *Anhang* bringt neben einigen für die Programmierarbeit nützlichen *Tabellen* die kompletten Befehlssätze der vier behandelten Universal-Mikroprozessoren in übersichtlicher Darstellung. Abgeschlossen wird das Buch mit einer Tabelle, aus der die wesentlichen Eigenschaften der heute auf dem Markt befindlichen *Lern- und Übungsgeräte* für die Mikroprozessor-Programmierung zu ersehen sind.

Behandelte Prozessortypen

Aus der Fülle der jetzt schon auf dem Markt befindlichen Mikroprozessoren werden vier in Deutschland häufig verwendete Typen, die man als universell verwendbar bezeichnen kann, herausgesucht, um Befehlsaufbau und Wirkungsweise ganz konkret zu beschreiben. Der sonst in der

Literatur vielfach gewählte Umweg, einen hypothetischen Mikroprozessor zu verwenden und mit einer abstrakten Befehlsbeschreibung zu arbeiten, wurde hier ausdrücklich nicht beschritten. Im einzelnen wurden die folgenden vier Prozessoren für dieses Buch ausgewählt:

2650 *(VALVO/SIGNETICS)* – ein Prozessor, der einen sehr vielseitigen und klar gegliederten Befehlssatz besitzt.

6502 *(MOS/ROCKWELL u.a.)* – ein moderner Prozessor, der in viele Lerngeräte und für den privaten Gebrauch bestimmte Computer eingebaut ist.

6800 *(MOTOROLA/FAIRCHILD u.a.)* – ein in Deutschland für die industrielle Anwendung weit verbreiteter Prozessor.

8080/85 *(INTEL/SIEMENS u.a.)* – als «Industriestandard» bekannt gewordener Prozessor, dessen Befehlssatz auch bei anderen Typen verwendet wird.

Nach dem Durcharbeiten dieses Buches sollte der Leser in der Lage sein, auch die Befehlssätze anderer Mikroprozessoren zu verstehen und die von den Herstellern herausgegebene beschreibende Literatur mit dem notwendigen Verständnis zu lesen.

Die im 6. Kapitel weiterhin noch vorgestellten Mikroprozessoren werden darum nur insoweit beschrieben, als sie wesentlich von den bisher behandelten Typen abweichen. Da die Hersteller von Mikroprozessoren laufend neue Typen herausbringen, die sich im Aufbau und im Befehlssatz an schon eingeführte Modelle anlehnen, wodurch eine durchaus gewünschte Familienähnlichkeit geschaffen wird, beschränkt sich dieses Kapitel auf die Behandlung der Typen

8048 *(INTEL, SIEMENS, VALVO u.a.)* als Beispiel für eine Typenfamilie von Ein-Chip-Prozessoren,

Z 80 *(ZILOG, MOSTEK u.a.)* als Beispiel für einen 8-Bit-Prozessor mit besonders umfangreichem Befehlssatz und

9900 *(TEXAS INSTRUMENTS)* als Beispiel für einen Prozessor mit 16-Bit-Architektur.

Sprache

Die Mikroprozessor-Technik stammt aus Amerika, und darum kommt auch niemand, der sich damit beschäftigen will, ohne eine gewisse Kenntnis der englischen Fachsprache aus. Für deutschsprachige Veröffentlichungen sollte dieses Fachamerikanisch aber nicht kommentarlos über-

nommen werden, denn dann sind diese nur für Fachleute lesbar. Es ist auch nicht zweckmäßig, alle Fachbegriffe unbedingt ins Deutsche zu übersetzen, da dies nur zur Sprachverwirrung beiträgt, weil jeder Autor dann seine eigenen Worte wählt.

In diesem Buch werden darum alle notwendigen Fachwörter, wenn sie zum ersten Mal vorkommen, in englischer Sprache als Versalien geschrieben und dann zusammen mit der deutschen Übersetzung erklärt. Soweit es sich um englische Fachausdrücke handelt, die inzwischen als Begriffe auch in die deutsche Sprache eingegangen sind, werden diese weiter verwendet. Überall dort, wo es eine vernünftige, kurze und eindeutige deutsche Übersetzung gibt, wird dieser der Vorzug gegeben. Um dem Leser jedoch die Einarbeitung in die Fachsprache zu erleichtern, werden aber auch bewußt deutsche und englische Begriffe nebeneinander gebraucht.

Leider gibt es nun auch im Amerikanischen neben den anerkannten Wörtern der Fachsprache noch viele andere, von den einzelnen Herstellerfirmen geprägte, oft sehr unterschiedliche Begriffe für gleiche Vorgänge oder Gegenstände.

Manchmal handelt es sich hierbei nur um Abkürzungen, die sich nur geringfügig unterscheiden und uns im allgemeinen wenig sagen. Viele dieser Ausdrücke werden in diesem Buch auch erläutert, dann aber nicht mehr verwendet, weil sie sich nicht allgemein durchgesetzt haben. Wer sich mit der amerikanischen Fachliteratur beschäftigt, muß deshalb leider damit rechnen, immer wieder auf neue Ausdrücke und Begriffe zu stoßen, die nicht mit Hilfe eines normalen Lexikons übersetzbar sind. Es ist darum durchaus empfehlenswert, bei der Programmierarbeit noch ein entsprechendes Speziallexikon zur Verfügung zu haben.

Benutzung des Buches

Den größten Nutzen von diesem Buch wird der Leser haben, der es nicht nur durchliest, sondern wirklich durcharbeitet. Bleistift und Papier sollten also immer neben dem Buch liegen.

Wichtig ist die Beherrschung des binären und des hexadezimalen Zahlensystems und deren Umcodierung in das Dezimalsystem, die immer wieder geübt werden sollte.

Die Beispiele sind in Assembler- und in Maschinensprache ausgedruckt. Als Lernbeispiele sind sie sehr ausführlich kommentiert, und es

werden für die verschiedenen Prozessoren meist unterschiedliche Lösungen vorgestellt. Zur Übung kann man zunächst nur den Assemblerteil betrachten und die Codierung in Maschinensprache mit Hilfe der Befehlstabellen selbst vornehmen.

Die vier genannten Prozessoren werden parallel behandelt. Textstellen, die nur für einzelne Prozessoren gelten, sind am Rand gekennzeichnet. Trotzdem ist es für Leser, die nur an einem Prozessor interessiert sind, empfehlenswert, auch den Text für die anderen Prozessoren zu lesen, denn gleiche Arbeitsweisen werden nicht jedesmal von neuem erklärt. Darum kann sich auch die Reihenfolge, in der die Programme und Befehle der einzelnen Prozessoren vorgestellt werden, von Abschnitt zu Abschnitt ändern.

Um die Arbeit der verschiedenen Prozessoren miteinander vergleichen zu können, sind die Programmbeispiele in mehreren Versionen für die verschiedenen Prozessoren geschrieben. Soweit Programme nur für einen Prozessortyp ausgedruckt sind, ist es auch wieder eine sehr nützliche Übung, die gleichen Programme für andere Typen zu schreiben.

Wenn am Anfang noch kein Lern- und Übungsgerät zur Verfügung steht, kann man die Lehrprogramme auch mit Bleistift und Papier schrittweise nachvollziehen, wie es im Kapitel 7 bei der Programmanalyse beschrieben ist. Wichtig ist es, wirklich zu verstehen, wie der Mikroprozessor im einzelnen Schritt für Schritt nach genau festgelegten Regeln arbeitet.

Um dieses Ziel zu erreichen, sollte man das Buch zunächst einmal durchlesen und dann aber beim zweiten Durchgang auch wirklich durcharbeiten.

1
Grundlagen für die Programmierarbeit

Dieses Buch wird sich in erster Linie nur mit der SOFTWARE, also den Programmen, die aus den in richtiger Weise miteinander verbundenen Bauelementen eines Mikroprozessor-Systems erst einen funktionierenden Computer machen, beschäftigen.

Anders als bei der Programmierung von Großcomputern, Homecomputern und auch programmierbaren Taschenrechnern, die man als BLACK BOX, also als schwarzen Kasten, dessen Inhalt nicht interessiert, verstehen kann, muß man bei der Arbeit mit Mikroprozessoren zumindest auch etwas über die HARDWARE, also über die elektronischen Bausteine des Systems, wissen.

1.1 Die Bauelemente eines Mikrocomputers

Grundsätzlich besteht jeder Computer aus einer *Zentraleinheit,* den *Eingabegeräten,* die es dem Menschen erlauben, mit dem Computer zu verkehren, und den *Ausgabemitteln,* mit deren Hilfe das Ergebnis der Arbeit des Computers zu erkennen ist (Bild 1.1). Der Programmierer eines Großcomputers weiß dann meist noch, daß die Zentraleinheit aus *Steuerwerk, Rechenwerk* und *Zentralspeicher* besteht, im übrigen wendet er bei der Programmierung ganz bestimmte Regeln an und kümmert sich nicht weiter um das Geschehen im Innern seines schwarzen Kastens. Weiterhin kennzeichnend für einen großen Computer ist es, daß er die verschiedensten Aufgaben je nach eingegebenem Programm durchführen kann.

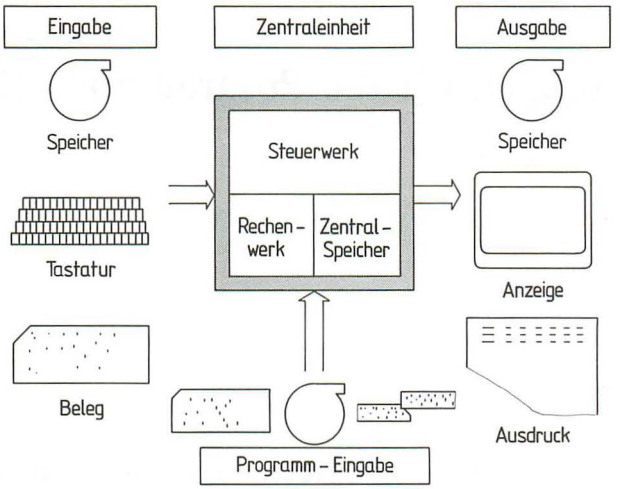


Bild 1.1 Hauptelemente eines Computers

Bei einem Computer kann man immer drei Hauptbaugruppen unterscheiden: Eingabeeinrichtungen, Zentraleinheit und Ausgabeeinrichtungen. Mit Hilfe der Eingabeeinheiten wird der Computer mit den zu verarbeitenden Daten gefüttert. Die Ausgabeeinheiten melden uns Menschen das Ergebnis der Datenverarbeitung. Die Zentraleinheit, in der die Datenverarbeitung stattfindet, wird von dem Programmierer meist als „Black Box" betrachtet, weil er sich um ihren Inhalt nicht zu kümmern braucht.

Ein großer Computer kann vom Benutzer beliebig für die verschiedensten Aufgaben programmiert werden. Das entsprechende Programm, auf Datenträgern (Lochkarten, Bandspeicher, Lochstreifen oder ähnlichem) gespeichert, wird vor der Benutzung in die Zentraleinheit eingegeben. Der Operateur hat dann nur noch die Geräte für Datenein- und -ausgabe zu bedienen.

Im Prinzip ist ein *Mikroprozessor-System,* also ein *Mikrocomputer,* ähnlich aufgebaut (Bild 1.2). Hier ist der Mikroprozessor das Herz des Systems und in gewisser Weise mit der Zentraleinheit zu vergleichen. Von ihm wird das System gesteuert, und in seinen elektronischen Schaltelementen finden alle Rechenvorgänge statt.

Ebenso gibt es auch beim Mikrocomputer *Ein- und Ausgabeeinheiten,* über die der Computer mit uns Menschen oder mit angeschlossenen Geräten verkehren kann. Ein wesentlicher Unterschied besteht jedoch in der Art, in der die Programme gespeichert werden.

Jedes Mikroprozessor-System hat zwei in ihrer Funktion unterschiedliche Speicherelemente, die mit den leicht verwechselbaren Namen ROM und RAM bezeichnet werden. Die Übersetzung aus dem Engli-

schen (ROM = READ ONLY MEMORY, *Nurlese-Speicher,* und RAM = RANDOM ACCESS MEMORY, *Speicher mit wahlfreiem Zugriff*) stellt den Unterschied nicht klar genug heraus. Die deutschen Ausdrücke *Festwert-Speicher* für ROM und *Schreib-Lese-Speicher* für RAM sind schon klarer, weil sie den wesentlichen Unterschied herausstellen. Verwendet werden aber trotzdem nur die englischen Abkürzungen, und die Verwirrung wird dadurch noch vergrößert, daß es auch noch PROM und EPROM gibt.

Grundsätzlich muß man wissen, daß der Inhalt in allen ROM, PROM und EPROM durch den Prozessor nicht verändert werden kann, aber auch nicht verlorengeht, wenn die Stromversorgung abgeschaltet wird. Diese Speicher sind also für die Aufnahme des Programms und von unveränderlichen Daten bestimmt. ROM können nur vom Hersteller in seinem Werk programmiert werden. PROM und EPROM kann auch der Anwender mit Hilfe eines kleinen Gerätes programmieren, wobei die gespeicherten Werte nur bei EPROM durch besondere Verfahren als Ganzes wieder gelöscht werden können.

Demgegenüber kann der Computer in RAM-Speicher während seiner Arbeit beliebig Daten und Informationen abspeichern und auch wieder

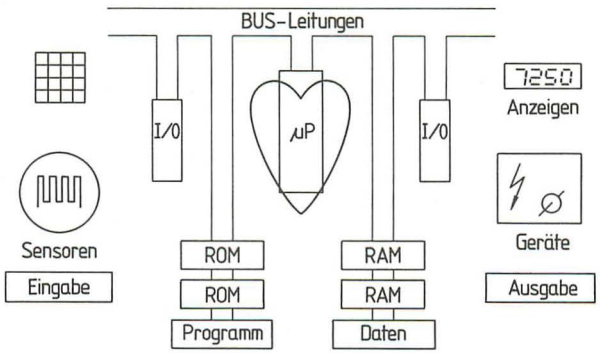


Bild 1.2 Aufbau eines Mikrocomputers
Bei einem Mikrocomputer ist der Mikroprozessor das Herzstück des Systems und mit der Zentraleinheit eines Großcomputers vergleichbar. Im Gegensatz zum Großcomputer ist das Programm bei einem Mikroprozessor für den Benutzer unzugänglich in einem elektronischen Speicher (ROM) enthalten. Im Rahmen dieses Programms können über Eingabe-/Ausgabeeinheiten (I/O UNITS) verschiedene Bauelemente und Geräte angeschlossen und bedient werden. RAM-Bausteine dienen zum Speichern von Daten während des Computerlaufs.

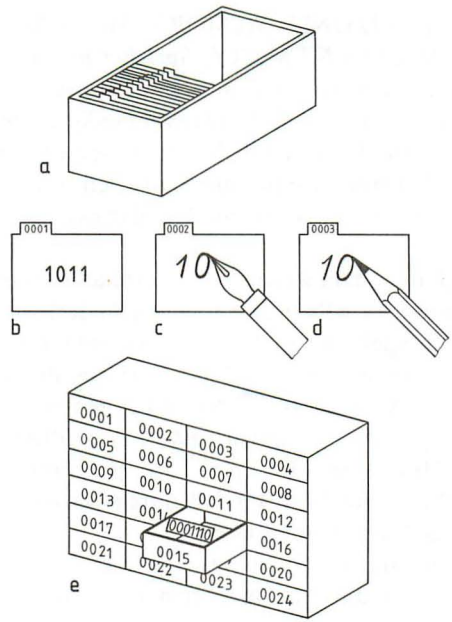


Bild 1.3 Die Funktionen der Speicherelemente in einem Mikrocomputer
Für die Bezeichnung der Speicher werden auch im Deutschen meist die leicht verwechselbaren englischen Abkürzungen verwendet. Man kann ihre Bedeutung am leichtesten an einem Beispiel erklären.

Ein Festwertspeicher ROM entspricht einer Kartei, in der die einzelnen Karten der Reihe nach geordnet und beziffert sind (a). Aufgrund der Ordnungsziffer, in der Computersprache Adresse genannt, kann man jede beliebige Karteikarte finden und den Inhalt lesen.

Die Karteikarten können bedruckt sein. Dies entspricht dem normalen ROM (b), bei dem der Speicherinhalt auch nur im Werk des Herstellers eingegeben werden kann. Die Karte kann aber auch bei der Anlage der Kartei, also beim Programmieren, beschrieben werden, und zwar unauslöschlich mit Tinte (c) entsprechend dem PROM oder mit Bleistift (d), so daß Ausradieren möglich ist, was einem EPROM entspricht.

Demgegenüber kann man sich einen Schreib-/Lesespeicher RAM als Schrank mit vielen Schubladen vorstellen (e). Jede Schublade trägt außen die Adreßnummer. Man kann beliebig viele Zettel mit Informationen, also mit Daten, in die Schubladen einlegen oder wieder herausholen, genau wie es elektronisch bei einem RAM möglich ist.

auslesen. Der gespeicherte Inhalt geht jedoch beim Abschalten der Spannung verloren. Die Funktion der verschiedenen Speicherelemente wird in Bild 1.3 anhand eines Modellbeispiels verdeutlicht.

Entscheidend wichtig für das Funktionieren eines Mikrocomputers ist

dann noch die Taktsteuerung und das BUS-System (Bild 1.4). Im Mikroprozessor werden die einzelnen Stromwege immer nur für eine ganz kurze Zeit geschaltet, wobei die Zeit, in der wirklich Strom fließt, durch den zu jedem Mikroprozessor-System gehörenden *Taktgeber* (CLOCK) bestimmt wird. Sendet dieser z.B. in jeder Mikrosekunde, also 1 Million mal pro Sekunde, sein Taktsignal aus, so können im Mikroprozessor in der Sekunde auch 1 Million verschiedene Schaltkombinationen aufgebaut werden. Bei jedem dieser Arbeitstakte wird, durch das Programm gesteuert, dann eine ganz bestimmte Aufgabe erfüllt.

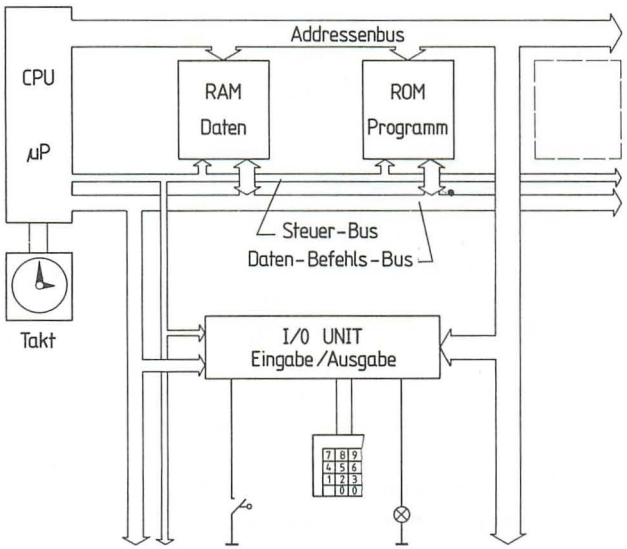


Bild 1.4 Bauelemente eines Mikroprozessor-Systems

Ein funktionsfähiges Mikroprozessor-System, also ein Mikrocomputer, benötigt außer dem eigentlichen Mikroprozessor-Baustein, der auch oft mit der englischen Abkürzung CPU (CENTRAL PROCESSING UNIT) bezeichnet wird, immer noch zusätzliche Bauelemente. Der Taktgeber, englisch CLOCK, steuert das ganze System in einer Folge von über einer Million Takten pro Sekunde.

Im Schreib-/Lesespeicher, englisch RAM (RANDOM ACCESS MEMORY), können Daten gespeichert und beliebig abgerufen werden. Der Festwertspeicher, englisch ROM (READ ONLY MEMORY), dient in erster Linie zum Speichern der Programmbefehle oder zum Speichern konstanter Daten. An Ein-/Ausgabeeinheiten, englisch I/O UNITS, werden periphere Geräte, über die der Mensch mit dem Mikroprozessor verkehren kann, angeschlossen.

Alle Bauelemente sind durch Datensammelleitungen, englisch BUS, miteinander verbunden.

Die Verbindung zwischen dem eigentlichen Mikroprozessor-Baustein, englisch auch CPU = CENTRAL PROCESSING UNIT genannt, und den anderen Bauelementen erfolgt durch *Datensammelleitungen,* englisch BUS. Bei den meisten Mikroprozessor-Systemen unterscheidet man zwischen einem *Adressenbus,* einem *Daten-/Befehlsbus* und verschiedenen Steuerleitungen, oft auch *Steuerbus* genannt. Die Zahl der jeweils in einem Bus parallel geführten Leitungen ist in den verschiedenen µP-Systemen unterschiedlich. Über die Funktion und Arbeitsweise der Busleitungen wird später noch etwas gesagt werden.

1.2 Verschiedene Systeme der Darstellung von Zahlen

Wenn man sich mit der Programmiertechnik befaßt, muß man die verschiedenen Möglichkeiten, eine bestimmte Menge darzustellen, kennen. Man spricht hier von verschiedenen *Zahlensystemen* oder richtiger eigentlich von verschiedenen *Zählsystemen.*

Das uns geläufige Dezimalsystem kennt zur Darstellung jeder beliebigen Mengen die zehn verschiedenen Ziffernsymbole **0, 1, 2, 3, 4, 5, 6, 7, 8** und **9.** Wenn die Menge größer ist als die Zahl der Symbole, wird eine neue Stelle angefangen, wobei grundsätzlich die Stelle mit der niedrigsten Wertigkeit rechts und die höchstwertigste Stelle links in der Ziffernfolge steht (Bild 1.5). Die darzustellende Gesamtmenge ergibt sich aus der Addition der Werte der Ziffern, jeweils multipliziert mit der Stellenwertigkeit, die wiederum als Potenz der Zahl der verwendeten Ziffernsymbole anzusetzen ist, so ergibt sich die auch in Bild 1.5 angegebene allgemeine Formel.

In der Digitaltechnik gibt es nur zwei genau definierte Zustände: **Ein – Aus** oder **Strom fließt – Strom fließt nicht** oder **Hohes Spannungsniveau – Niedriges Spannungsniveau.** Um diese Zustände zu beschreiben, werden also auch nur zwei Symbole benötigt, und so liegt es nahe, daß auch alle Computer im *binären Zahlensystem,* das auch nur zwei Ziffernsymbole kennt, arbeiten. Dieses System der *Dualzahlen* verwendet nur die Ziffernsymbole **0** und **1.** Es werden darum wesentlich mehr Stellen als beim Dezimalsystem benötigt, denn nach der erwähnten Formel kann mit der niederstwertigen Stelle nur die Mengen 0 und 1 ausgedrückt werden. Bei zwei Stellen beträgt der maximale Informations-

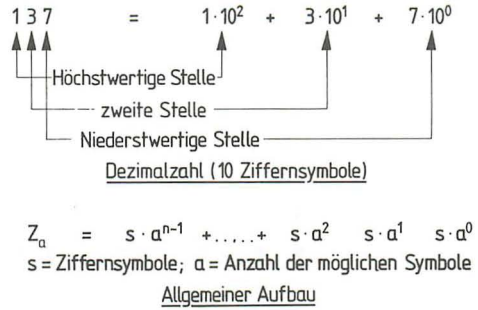


Bild 1.5 Grundsätzlicher Aufbau der Ziffern in einem Zahlensystem
Jedes Zahlensystem verwendet eine begrenzte Anzahl von Ziffernsymbolen, die beim Aufwärtszählen nacheinander eingesetzt werden. Ist die Anzahl der Ziffernsymbole erschöpft, wird eine neue Stelle links neben der bisherigen Zählstelle begonnen, in der jedes Ziffernsymbol einen Wert hat, der dem Produkt aus Symbolwert und Stellenwert nach der oben aufgeführten Formel entspricht.

inhalt 2^2, dann 2^3 usw. Es muß immer eine neue Stelle begonnen werden, wenn die Kombinationsmöglichkeit von 0 und 1 in den bisherigen Stellen erschöpft ist (siehe auch Bild 1.7).

Für uns Menschen ist es nun sehr schwer, die computergerechten Binärzahlen zu behalten oder gar zu verarbeiten. Schon die Aussprache einer vierstelligen Binärzahl macht Schwierigkeiten. So verwendet man in der Programmiertechnik, allein um dem Menschen das Verständnis zu erleichtern, noch zwei andere Zahlensysteme:

das *Oktalsystem* und das *Hexadezimalsystem.*

Das *Oktalsystem* arbeitet mit den 8 Ziffernsymbolen **0, 1, 2, 3, 4, 5, 6** und **7.** Mit einem dieser Symbole kann man immer drei Stellen einer Binärzahl darstellen. Eine 1 in der letzten Binärstelle hat den Wert $2^0 = 1$. Die zweite Stelle entspricht $2^1 = 2$ und die dritte $2^2 = 4$, so daß der Gesamtwert von 3 Binärstellen höchstens 7 sein kann und somit durch eine Oktalstelle auszudrücken ist.

In ähnlicher Weise ist es möglich, 4 Binärstellen durch eine *Hexadezimalzahl* zu ersetzen. Man braucht dafür dann allerdings 16 verschiedene Ziffernsymbole, und da wir nur 10 Ziffern kennen, werden die Buchstaben **A, B, C, D, E** und **F** als Ziffernsymbole mit hinzugenommen.

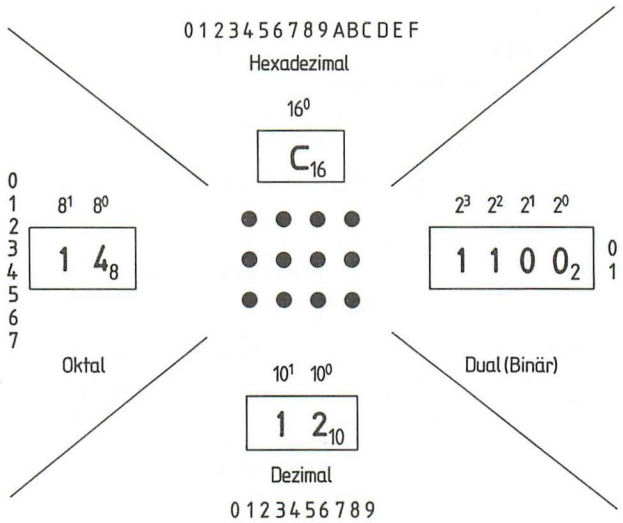


Bild 1.6 Darstellung einer Menge in verschiedenen Zahlensystemen
Die in dem Punktraster in der Mitte dargestellte Menge wird in den vier gezeigten Zahlensystemen, die mit unterschiedlicher Anzahl von Symbolen arbeiten, völlig verschieden geschrieben.

In Bild 1.6 wird gezeigt, wie eine bestimmte Menge in den vier besprochenen Zahlensystemen dargestellt werden kan. Bild 1.7 zeigt eine Tabelle mit der Darstellung jeweils der gleichen Zahl in verschiedenen Systemen.

> Bei allen Zahlensystemen muß man daran denken, daß die Ziffer **0** auch ein definierter Wert ist. In der Elektronik beginnen Zählungen darum auch meist schon mit dem Wert **0** und nicht wie sonst üblich mit dem Wert **1.**

Bei der Programmierung von Mikroprozessoren hat sich die hexadezimale Darstellung von Binärzahlen durchgesetzt, da diese der inneren Organisation der Prozessoren am besten entspricht. In den folgenden Kapiteln wird darum auch nur noch diese Darstellung angewandt, und es

Dezimal 10^n	Dual (Bimär) 2^n	Oktal 8^n	Hex 16^n
3 2 1 0	9 8 7 6 5 4 3 2 1 0	3 2 1 0	2 1 0
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
31	11111	37	1F
64	1000000	100	40
208	11010000	320	D0
255	11111111	377	FF
256	100000000	400	100
1000	1111101000	1750	3E8

Bild 1.7 Zahlendarstellung in verschiedenen Systemen

wird empfohlen, die Umcodierung von hexadezimal in binär und umgekehrt an selbstgewählten Beispielen ausreichend zu üben.

Da in Schriften und Veröffentlichungen, aber auch in Programmen oft Zahlen in verschiedenen Systemen nebeneinander erscheinen, ist es notwendig, dafür zu sorgen, daß verwechselbare Zifferndarstellungen eindeutig dem gemeinten Zahlensystem zugeordnet werden. In wissenschaftlicher Schreibweise bedient man sich der Indizes, die jeweils die Basis des Systems, also die Anzahl der verwendeten, verschiedenen Ziffernsymbole, angibt. So bedeutet z.B. $\mathbf{101}_{10}$ auch die Dezimalzahl **101.** Dagegen hat $\mathbf{101}_{2}$ den dezimalen Wert **5.** Dementsprechend ist $\mathbf{101}_{8}$ mit dezimal **65** und $\mathbf{101}_{16}$ mit dezimal **257** gleichzusetzen.

Vielfach werden auch die Buchstaben **B** für binär, **D** für dezimal, **O** oder **Q** für oktal und **H** für hexadezimal verwendet. Leider gibt es dafür aber keine einheitlichen Regeln, da die gleiche Hexadezimalzahl bei der

Dezimal				BCD											
10^3	10^2	10^1	10^0	10^3				10^2				10^1				10^0			
				2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
			0													0	0	0	0
			1													0	0	0	1
			2													0	0	1	0
			3													0	0	1	1
			4													0	1	0	0
			5													0	1	0	1
			6													0	1	1	0
			7													0	1	1	1
			8													1	0	0	0
			9													1	0	0	1
		1	0									0	0	0	1	0	0	0	0
		1	5									0	0	0	1	0	1	0	1
		4	5									0	1	0	0	0	1	0	1
	1	0	0					0	0	0	1	0	0	0	0	0	0	0	0
	1	4	5					0	0	0	1	0	1	0	0	0	1	0	1
	7	9	9					0	1	1	1	1	0	0	1	1	0	0	1
	9	9	9					1	0	0	1	1	0	0	1	1	0	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	4	5	0	0	1	0	0	0	0	1	0	1	0	0	0	1	0	1
4	5	1	7	0	1	0	0	0	1	0	1	0	0	0	1	0	1	1	1
7	7	7	6	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	0
8	3	3	8	1	0	0	0	0	0	1	1	0	0	1	1	1	0	0	0
8	7	6	5	1	0	0	0	0	1	1	1	0	1	1	0	0	1	0	1
9	1	2	3	1	0	0	1	0	0	0	1	0	0	1	0	0	0	1	1
9	9	9	9	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1

Bild 1.8 Dezimalzahlen im Vergleich zu gleichwertigen Zahlen im BCD-System

Programmierung je nach Hersteller einmal **H'7F3C'** und zum anderen **7F3CH** geschrieben werden muß. Bei einigen Prozessoren dient das vorgestellte Dollar-Zeichen ($) als Kennzeichen für Hexadezimalzahlen.

Nun gibt es noch eine Darstellung für Zahlen, die für die Mikroprozessor-Technik Bedeutung hat, bei der aber nicht das oben erklärte allgemeine Bildungsgesetz anzuwenden ist. Man spricht hier von *BCD-Zahlen,* was eine Abkürzung für BINARY CODED DECIMAL ist. Bei diesem System werden nur die ersten 10 Kombinationen von **0** und **1,** die vier Binärzahlen zulassen, ausgenutzt. Sie entsprechen dann den Ziffern **0** bis **9** des Dezimalsystems. Immer wenn dezimal eine neue Stelle angefangen wird, muß auch hier ein neuer Viererblock von Binärziffern, den man auch *Tetrade* nennt, begonnen werden (Bild 1.8).

Der Vorteil besteht darin, daß BCD-Zahlen leicht im Kopf in Dezimalzahlen umzurechnen sind. Nachteilig ist es, daß ein Teil der Infor-

mationsmöglichkeiten, die 4 Binärziffern enthalten können, nicht ausgenutzt wird. Außerdem ist das Rechnen mit BCD-Zahlen etwas umständlich.

1.3 Das Programm als Folge von Befehlen

Jeder Mikroprozessor hat seinen ganz bestimmten *Befehlssatz* und kann dementsprechend eine begrenzte Anzahl von Befehlen ausführen. Die Art und Reihenfolge der auszuführenden Befehle wird durch das *Programm* bestimmt. *Grundsätzlich kann immer nur ein Befehl zur Zeit verarbeitet werden.*

> Ein Mikroprozessor-Programm muß also alle Befehle, die zur Erreichung eines gewünschten Zwecks notwendig sind, in der richtigen Folge in einer dem Prozessor verständlichen Sprache enthalten.

Grundsätzlich versteht ein Mikroprozessor nur seine *Maschinensprache,* die aus hohen und niedrigen Spannungsniveaus, die ihm auf den Leitungen seines Befehlsbus zu einem durch seinen Takt definierten Zeitpunkt angeboten werden müssen, besteht. Diese Sprache kann als binäre Information betrachtet werden, die zweckmäßigerweise in Hexadezimalzahlen geschrieben wird, um für uns Menschen besser handhabbar zu sein.

Die kleinste so darstellbare Informationsmenge, die aus den Zuständen **0** und **1** besteht, wird *Bit,* abgeleitet von BINARY DIGIT, genannt. Während 1 Bit zwei verschiedene Informationen enthalten kann, können in 4 Binärstellen, also in 4 Bit, bereits 16 verschiedene Kombinationen von **0** und **1,** das heißt 16 verschiedenen Informationen, untergebracht werden.

Mit der Zahl der Stellen erhöht sich der mögliche Informationsinhalt, wie in Bild 1.9 gezeigt wird. Ein Bitmuster mit 8 Bit wird *Byte* genannt und ergibt 256 verschiedene Informationsinhalte. Für 4 Bit setzt sich immer mehr der Ausdruck *Nibble* und für 16 Bit die Bezeichnung *Wort* durch.

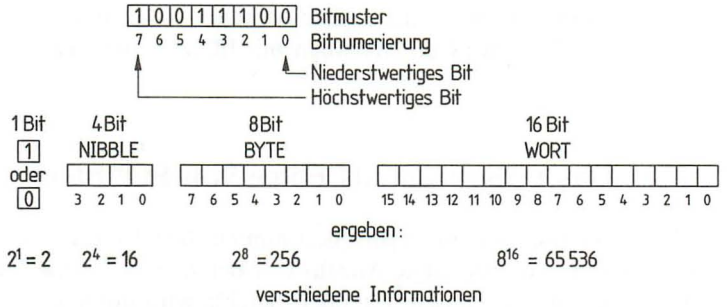


Bild 1.9 Verschiedene Bitmuster, Bitnumerierung und Informationsinhalt

Der Informationsinhalt eines Bitmusters, d.h. die Zahl der verschiedenen Informationen, die durch Kombinationen von **0** und **1** darstellbar sind, entspricht immer den Werten der Potenzen der Zahl 2 mit der Anzahl der Bitpositionen, die betrachtet werden sollen als Exponent. Die Bitnumerierung mit **0** rechts beginnend und gewisse Ausdrücke für bestimmte Bitmusterlängen haben sich eingebürgert.

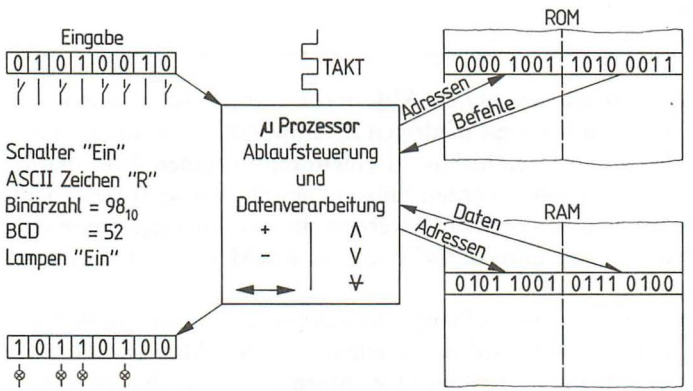


Bild 1.10 Funktionsablauf in einem Mikroprozessor-System

Eine Information, die als Bitmuster über die Eingabeeinheit an den Mikroprozessor gegeben wird, kann völlig beliebig interpretiert werden. Das gleiche Bitmuster kann so z.B. die Stellung von Schaltern oder einen Buchstaben nach dem ASCII-Code oder auch eine Binärzahl oder eine BCD-Zahl bedeuten.

Diese Information wird im Prozessor nach gewissen Regeln, die er sich als Befehle einen nach dem anderen aus dem ROM-Speicher holt, verarbeitet. Dabei benötigte Daten kann er aus einem RAM-Speicher holen oder dort abspeichern.

Das Ergebnis der Verarbeitung wird dann wieder als Bitmuster ausgegeben und kann in beliebiger Weise, z.B. für das Einschalten von Lampen, interpretiert werden.

Die Zählung der Bit in einem Bitmuster beginnt im allgemeinen mit dem niederstwertigen Bit, das rechts steht und den Stellenwert **0** hat und dementsprechend auch den Zählwert **0** erhält.

Wie ein Mikroprozessor-System im Prinzip arbeitet, ist in Bild 1.10 dargestellt. Ziel jeder Computertätigkeit ist es, eingegebene Informationen in irgendeiner Weise zu verarbeiten und dann die Ergebnisse dieser Verarbeitung wieder auszugeben.

Die Eingabe der Information erfolgt als *Bitmuster,* dem durch den Programmierer eine beliebige Bedeutung zugewiesen werden kann. Gewisse Interpretationen sind üblich. So kann das Bitmuster z.B. als Binärzahl oder in der internationalen ASCII-Codierung auch als Buchstabe verstanden werden. Ebenso kann ein Bitmuster aber auch die Schaltzustände von Schaltern oder Lampen darstellen.

Damit der Prozessor die Eingabeinformationen hereinholen kann, braucht er entsprechende Anweisungen, die in den Befehlen des Programms stehen. Diese Befehle sind wiederum auch als Binärinformationen in einem ROM-Speicher abgelegt. Der Prozessor ist in der Lage, diese Befehle nacheinander in einer auch wieder durch das Programm festgelegten Reihenfolge zu lesen, zu interpretieren und auszuführen.

Je nach Befehl holt er Eingabeinformationen und/oder im RAM oder ROM gespeicherte Daten und verarbeitet diese, indem sie durch verschiedene Operationen miteinander verknüpft werden. Es entstehen dadurch neue Bitmuster, also veränderte Informationen, die entweder in einem RAM zwischengespeichert oder als Ergebnisse der Verarbeitung ausgegeben werden. Die richtige Interpretation des ausgegebenen Bitmusters bringt dann den Erfolg der Arbeit.

1.4 Befehlsablauf und Speicherorganisation

In einem Mikroprozessor-System bestehen während einer durch ein Taktsignal bestimmten sehr kurzen Zeit ganz bestimmte Spannungszustände auf den einzelnen Busleitungen und auch in gewissen Speicherzellen und Bauelementen. Hierdurch werden kurzzeitig Schaltwege geschaffen, durch die Stromimpulse fließen können.

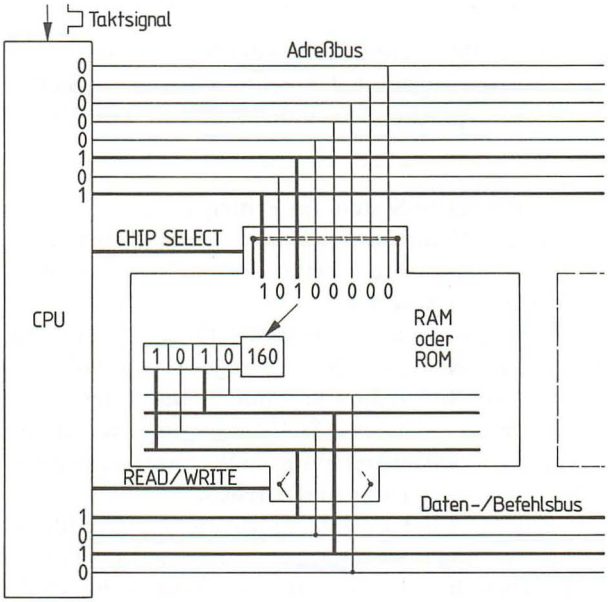


Bild 1.11 Funktionen der BUS-Systeme in einem Mikrocomputer
Während einer bestimmten Taktperiode steht am Adressenausgang des Mikroprozessors, also der Zentraleinheit des Mikrocomputers, ein ganz bestimmtes Bitmuster in Form von hohen und niedrigen Spannungspegeln an. Jede an den Adressenbus angeschlossene Einheit hat eine eigene Decodierschaltung, die dafür sorgt, daß nur eine einzige Speicherzelle, deren Adresse dem Bitmuster des Adressenbus entspricht, leitend mit dem zweiten Bus verbunden wird. Je nach dem Zustand der Steuerleitung, die bildlich der Wirkung eines Tors entspricht, wird dann von der Zentraleinheit ein Wert in den Speicher eingeschrieben oder ausgelesen. Eine weitere Steuereinheit am Adresseneingang kann ein weiteres Tor öffnen und schließen, so daß immer nur eine ganz bestimmte Speichereinheit angesprochen wird.

In Bild 1.11 ist ein kurzzeitiger Zustand einiger Busleitungen dargestellt. Auf dem obenliegenden Adreß-Bus haben die erste und die dritte Leitung ein hohes und alle anderen ein niedriges Spannungsniveau. Das entspricht dem Bitmuster **10100000**. Dieses Bitmuster liegt also auch an den Adreßeingängen aller im System angeschlossenen Speicherelemente an.

Nun besitzt jeder Speicher eine sogenannte *Decodierschaltung*, die bewirkt, daß nur die Speicherstelle, die durch das Bitmuster der Adresse

angesprochen wird, elektrisch mit dem zweiten Bus, dem Daten- und Befehlsbus, verbunden ist. Die angesprochene Speicherstelle ist so aufgebaut, daß sie bei einem Prozessor, der 8 Bit parallel verarbeiten kann, auch 8-Bit-Information speichern kann. Im Bild 1.11 ist zur Vereinfachung ein 4-Bit-Prozessor dargestellt, der also auch nur 4 Leitungen für den Daten-Befehlsbus braucht. Grundsätzlich wird die gespeicherte Information parallel in die CPU übertragen, so daß ein 8-Bit-Prozessor hier auch 8 Leitungen hat.

Handelt es sich bei dem Speicher um ein ROM, so können nur Informationen ausgelesen werden. Der Datentransport geht also immer vom Speicher zur CPU. Wird dagegen eine Speicherstelle in einem RAM adressiert, muß der Prozessor auch noch bestimmen, ob eine Information *gelesen* (READ), d.h. vom Speicher übernommen, oder in diesen *eingeschrieben* (WRITE), d.h. dort abgespeichert werden soll. Hierzu dienen die Steuerleitungen, die vom Prozessor auch auf hohes oder niedriges Niveau geschaltet werden können.

Andere Steuerleitungen dienen dazu, ganz bestimmte Bausteine anzuwählen. Alle Speicherelemente haben am Adreßeingang eine sogenannte *Torschaltung,* die den Eingang nur dann auf den Adreßbus schaltet, wenn die dazugehörige Steuerleitung auf dem vereinbarten Niveau, also hoch oder niedrig, liegt. Auf englisch werden diese Leitungen CHIP SELECT oder CHIP ENABLE genannt. Oft werden einzelne Leitungen des Adreßbus hierfür verwendet. Hierdurch wird vermieden, daß jeder an den Bus angeschlossene Baustein Decodierschaltungen für alle Adreßleitungen haben muß.

Um dies zu verstehen, muß man den Aufbau des gesamten *Adreßfeldes,* das ein Mikroprozessor bedienen kann, betrachten (Bild 1.12). Im allgemeinen haben die 8-Bit-Prozessoren 16 Adreßleitungen, die mit **A0** bis **A15** bezeichnet werden. Hiermit lassen sich einschließlich der Adresse **0** insgesamt **65 536** verschiedene Adressen ansprechen.

Die erste Adresse, also die Adresse **0,** wird angewählt, wenn alle Leitungen niedriges Spannungsniveau haben, und die höchste Adresse, wenn alle Leitungen auf hohes Niveau geschaltet sind. Entsprechend dem Bitmuster der Adressenleitungen werden die Adressen in hexadezimaler Codierung mit **0000** bis **FFFF** bezeichnet.

Um die *Speicherkapazität* eines Bausteins anzugeben, verwendet man gern die Abkürzung **K,** die eigentlich Kilo, also Tausend, bedeutet. In der Elektronik bedeutet 1 K jedoch 1024 Speicherstellen. Etwas undurch-

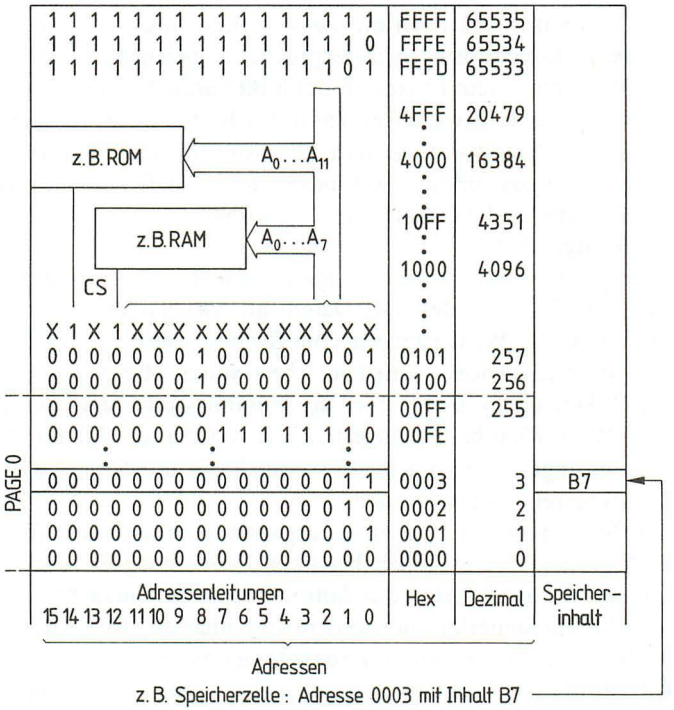


Bild 1.12 Speicherorganisation für einen Mikrocomputer

Die meisten 8-Bit-Mikroprozessoren haben einen Adreßbus mit 16 Leitungen, die mit A 0 bis A 15 bezeichnet werden, Mit ihnen lassen sich 65 536 verschiedene Adressen anrufen, was der Hexadezimalzahl **FFFF** entspricht. Der gesamte ansprechbare Adreßraum läßt sich also auch durch eine Hexadezimalzahl mit vier Stellen darstellen.

Oft wird eine Adreßleitung als Anwahlleitung (CHIP SELECT bzw. CHIP ENABLE) geschaltet. Ein auf dieser Leitung anstehendes Signal aktiviert den Baustein, und je nach seiner Kapazität erfolgt die eigentliche Adressierung dann durch einige der unteren Adreßleitungen.

sichtig wird diese Bezeichnung, weil manchmal die Zahl der einzelnen Speicherzellen, in denen jeweils nur 1 Bit gespeichert werden kann und andererseits wiederum die Zahl der speicherbaren Byte genannt wird. Man sollte darum, wenn Speicherkapazitäten genannt werden, immer hinzufügen, ob Bit oder Byte gemeint sind. Dies kann auch in der Form: 1 K × 8 oder 8 K × 1 erfolgen, was zugleich auch die Organisation des Speichers angibt.

Wenn man nun nicht den ganzen Adressierbereich des Mikroprozessors, der 64 KByte beträgt, benötigt, kann man jeweils eine Adreßleitung als *Bausteinanwahlleitung* (CHIP SELECT) schalten, wie es in Bild 1.12 dargestellt ist. Das eingezeichnete RAM hat eine Kapazität von 256 Speicherplätzen, also von $^1/_4$ KByte. Zur Darstellung dieser Adreßmenge reichen 8 Leitungen. Wenn man nun hierfür z.B. die Leitungen A_0 bis A_7 verwendet, genügt dies. Zusätzlich muß dann noch eine weitere Adreßleitung zur Bausteinanwahl verwendet werden, im Beispiel ist dies die Leitung A_{12}. Der Adreßbereich für diese RAM liegt damit zwischen hex **1000** und **10FF.**

In gleicher Weise ist das als Beispiel gezeichnete ROM angeschlossen. Als Anwahlleitung dient A_{14}. Der Baustein hat eine Kapazität von 4 KByte, wofür 12 Adreßleitungen genügen.

Bei einigen Mikroprozessoren wird der Adressierbereich noch in einzelne *Seiten* (PAGE) aufgeteilt. In diesem Fall kennzeichnet der Wert des oder der höchstwertigen Bit der Adresse die Seite. Es ist auch üblich, den unteren Speicherbereich, der mit nur 8 Bit adressiert werden kann, als *Seite* **0** (PAGE ZERO) zu bezeichnen.

1.5 Takt, Maschinenzyklus und Befehlsausführungszeit

Die Zeit, die ein Mikroprozessor zur Ausführung eines Befehls braucht, liegt im Mikrosekundenbereich und ist somit außerordentlich kurz. Mit unseren menschlichen Sinnen können wir diese kurzen Zeiten nicht erfassen. Sie sind höchstens an einem Beispiel erklärbar (Bild 1.13).

Wenn angenommen wird, daß ein Mikroprozessor im Durchschnitt 10 Mikrosekunden für einen Befehl braucht, kann er in einer Sekunde 100 000 Befehle ausführen. Dies ist eine sehr hohe Leistung, und im allgemeinen wird man bei der Verwendung von Mikroprozessoren für

Sekunde	m = Milli	μ = Mikro	n = Nano
1	0,001	0,000 001	0,000 000 001
1 Sekunde	1 tausendstel Sekunde	1 millionstel Sekunde	1 milliardstel Sekunde

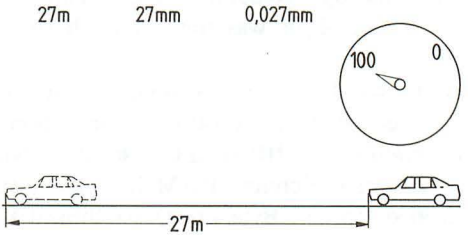


Bild 1.13 Die Unterteilung der Sekunde

Die kurzen Zeiten, in denen sich die Schaltvorgänge und Befehlsabläufe in einem Mikroprozessor abspielen, sind für die menschlichen Sinne nicht erfaßbar. Sie können nur an einem Beispiel erklärt werden: Ein Auto, das mit 100 km/h fährt, legt in der Sekunde ca. 27 m zurück. In einer Millisekunde fährt es noch 27 Millimeter und in der Mikrosekunde nur noch 0,027 Millimeter, was der Hälfte der Dicke eines Haares entspricht.

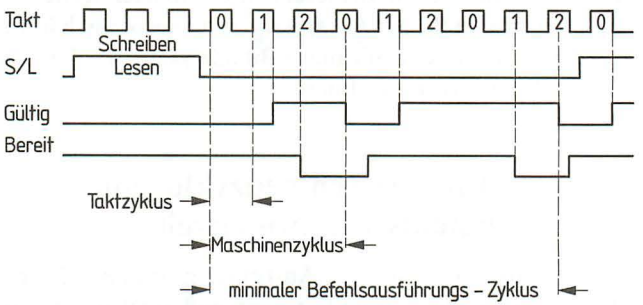


Bild 1.14 Taktzyklen in einem Mikroprozessor-System

Der zu jedem Mikroprozessor-System gehörige Taktgeber (CLOCK) arbeitet meist mit einer Frequenz von mehreren Megahertz, erzeugt also in der Sekunde mehrere Millionen Schwingungen. Eine vollständige Schwingung ist ein Taktzyklus. Der Mikroprozessor braucht für die kleinste mögliche Operation immer mehrere Taktzyklen, woraus sich der Maschinenzyklus ergibt. Die meisten Befehle wiederum benötigen mehrere Maschinenzyklen, die dann die Befehlsausführungszeit bestimmen.

die üblichen Steuerungs- und Rechenaufgaben keine Zeitprobleme haben. Trotzdem muß man beim Programmieren auch immer an die Laufzeit bestimmter Programmabschnitte denken und diese gegebenenfalls berechnen.

Hierbei muß man nun zwischen dem Takt, den der Taktgeber erzeugt, dem Maschinenzyklus, in dem der Prozessor arbeitet, und der Ausführungszeit für einen Befehl unterscheiden. Leider wird im Englischen das Wort CYCLE mit verschiedener Bedeutung verwendet, so daß man genau feststellen muß, ob ein CLOCK CYCLE, also eine komplette *Periode des Taktsignals,* ein MACHINE CYCLE, also ein *Maschinenzyklus,* oder ein INSTRUCTION CYCLE, also die *Ausführungszeit für einen Befehl,* gemeint ist.

Grundsätzlich ist die *Taktperiode,* englisch manchmal auch STATE genannt, die kleinste Zeit. Ein Mikroprozessor braucht immer mehrere Takte, um eine Teilaufgabe innerhalb eines Befehls auszuführen. Man nennt diese Zeit darum meist Maschinenzyklus, denn ein Maschinenzyklus ist die kleinstmögliche Zeit für die Abarbeitung einer einfachen *Operation.* Da in den meisten Befehlen mehrere Prozessor-Operationen zusammengefaßt sind, dauert die Befehlsausführungszeit meist mehrere Maschinenzyklen (Bild 1.14).

Während der Takt den Zeitmaßstab setzt, bestimmen die Verhältnisse auf den Steuerleitungen, was während der einzelnen Taktperioden geschieht. Die einzelnen Mikroprozessor-Typen unterscheiden sich bezüglich der Steuerleitungen sehr stark. Grundsätzlich gibt es aber immer eine Leitung, durch die der Prozessor anzeigt, daß die von ihm auf den Busleitungen ausgegebenen Informationen gültig sind, und eine andere Leitung, mit der die angeschlossenen Speicher die Bereitschaft zur Übernahme oder Abgabe von Informationen melden. Eine dritte Leitung bestimmt dann meist, ob geschrieben, also eine Information abgespeichert, oder gelesen, das heißt eine Information in die CPU übernommen werden soll.

Ein vollständiger Befehlszyklus läuft meist wie folgt ab:

- Der Prozessor gibt auf dem Adreßbus die Adresse des nächsten Befehls aus, zusammen mit den Steuersignalen *«Gültig»* und *«Lesen».*
- Wenn der Speicher *«bereit»* ist, gelangt der Inhalt der angesprochenen Adresse auf den Datenbus und damit in den Prozessor.

- Der Prozessor erkennt die eingegangene Information als Befehl und interpretiert sie.
- Wenn alle für die Befehlsausführung notwendigen Daten bereits in der CPU stehen, wird der Befehl ausgeführt.
- Wenn für die Befehlsausführung noch Daten benötigt werden, wird deren Adresse auf den Adreßbus gegeben und die Daten aus der angesprochenen Speicherstelle in die CPU eingelesen.
- Wenn durch einen Befehl die Abspeicherung von Daten bestimmt wurde, wird die Adresse des Speicherplatzes für diese Daten zusammen mit den Steuerbefehlen *«Schreiben»* und *«Gültig»* ausgegeben.
- Nach Eingang des *«Bereit»*-Signals kann dann ein neuer Befehlszyklus beginnen.

1.6 Struktureller Aufbau eines Mikroprozessors

Wenn man ein Programm für einen Mikrocomputer schreiben will, muß man etwas über den inneren Aufbau des darin verwendeten Mikroprozessors wissen. Jeder Prozessor besitzt eine Anzahl von *schnellen Speichern,* die man hier im allgemeinen *Register* nennt, auf die durch Programmbefehle unmittelbar zugegriffen werden kann. Darüber hinaus gibt es weitere *Hilfsregister,* um die sich der Programmierer aber nur zum Teil kümmern muß. Die einzelnen Typen der verschiedenen Hersteller unterscheiden sich im inneren Aufbau beträchtlich. Es gibt aber eine Reihe von Funktionselementen, die in allen Prozessoren anzutreffen sind, sie sind im Schema in Bild 1.15 dargestellt.

Jeder Mikroprozessor besitzt einen *Befehlsadreß-Speicher,* manchmal auch *Programm-Zähler,* PROGRAM COUNTER (PC), genannt. Er enthält grundsätzlich immer die Adresse, in der das nächste zu verarbeitende Befehlsbyte zu finden ist. Normalerweise wird er jeweils nach Verarbeitung eines Befehlsbyte um den Wert **1** erhöht, man nennt dies auch *Incrementierung.* Durch gewisse Programmbefehle kann der Befehlsadreß-Speicher auch mit einer beliebigen Adresse geladen werden.

In jedem Prozessor gibt es dann ein oder mehrere durch Befehle vielseitig ansprechbare *Hauptregister,* die man *Akkumulator* nennt. Über dieses oder diese Register laufen alle Datentransport- und Datenverarbeitungsvorgänge.

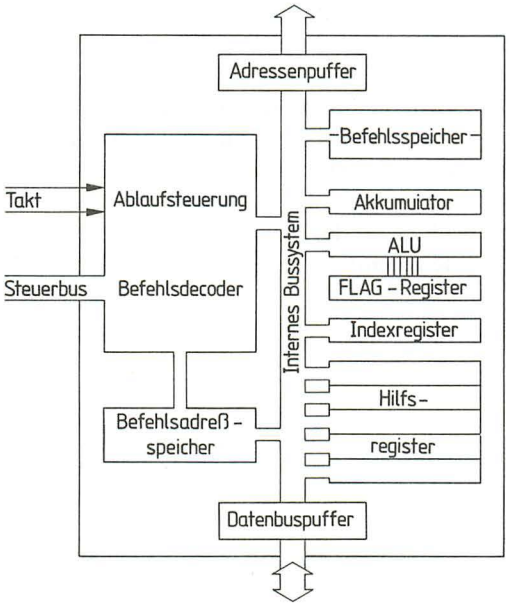


Bild 1.15 Grundsätzlicher Aufbau eines Mikroprozessors
Die Mikroprozessoren verschiedener Hersteller sind recht unterschiedlich aufgebaut. Es gibt jedoch eine Reihe von Funktionselementen, die in allen Typen vorkommen.
Jeder Mikroprozessor besitzt eine Reihe von schnellarbeitenden Speichern, die Register genannt werden. Sie können über Programmbefehle angesprochen werden und speichern je nach Aufgabe Adressen, Befehle oder Daten. Es gibt meist ein (oder mehrere) Hauptregister, Akkumulator genannt, über das der Datenverkehr mit den Speicherelementen erfolgt. Eine Arithmetik-Logik-Einheit, ALU genannt, dient der eigentlichen Datenverarbeitung. Der gesamte Ablauf bei der Ausführung eines Befehls wird durch die im Takt arbeitende Ablaufsteuerung in Verbindung mit dem Befehlsdekoder gesteuert.

Im engen Zusammenhang mit dem Akkumulator, oft *Akku* genannt, arbeitet die sogenannte *Arithmetik-Logik-Einheit,* englisch *ALU* = ARITHMETIC-LOGIC-UNIT, in der die eigentliche Datenverarbeitung stattfindet.

Die Ergebnisse dieser Verarbeitung werden dann in einem weiteren Register, das *FLAG-Register* oder *Programm-Statur-Register* genannt wird, als Information eingeschrieben.

Es gibt dann noch immer eine Reihe von Hilfsregistern, die der Pro-

zessor für seine Arbeit braucht. Bei einigen Prozessortypen können verschiedene Hilfsregister auch durch Programmbefehle angesprochen werden.

Es ist üblich, ein oder mehrere Hilfsregister auch als *Indexregister* auszustatten. Mit ihrer Hilfe kann man im Programmablauf Adressen modifizieren, ohne dafür jedesmal einen neuen Befehl verwenden zu müssen, wie es noch später erklärt werden wird.

Der gesamte Ablauf bei der Ausführung eines Befehls wird durch die im Takt arbeitende *Ablaufsteuerung* zusammen mit dem *Befehlsdecoder* gesteuert.

2
Etwas Binär-Mathematik

Alle Mikroprozessoren arbeiten grundsätzlich rein binär. Dies bedeutet, daß für alle Datenverarbeitungsvorgänge, also auch für alle mathematischen Operationen, die Regeln der binären Mathematik zur Anwendung kommen.

Wenn man sich mit der Mikroprozessor-Programmierung beschäftigen will, ist es unabdingbar, daß man zumindest die wichtigsten Regeln der Binär-Mathematik kennt. Man braucht dabei aber keinesfalls bemüht zu sein, sich selbst größere Rechenfertigkeit in diesem Zahlensystem zuzulegen, denn das Rechnen selbst macht der Mikroprozessor viel schneller, als wir es mit unserem Gehirn vermögen.

Es kommt hierbei nur auf das Verständnis und nicht auf das Beherrschen der Rechenmethoden an.

Es ist darum auch nicht notwendig, Programme für die verschiedenen Rechenverfahren, die ein Mikroprozessor ausführen kann, selbst zu entwerfen, denn diese kann man sich in Form sogenannter *Standardroutinen* meist ohne Schwierigkeiten von den Herstellern der Mikroprozessoren beschaffen.

Es genügt darum, wenn Sie dieses Kapitel zunächst einmal durchlesen, damit Sie wissen, wie ein Mikroprozessor bei den verschiedenen Rechenarten arbeitet. Obwohl ein Mikroprozessor nur wenige Grundoperationen ausführen kann, ist es möglich, durch geschicktes Aneinanderreihen von Befehlen praktisch alle nur denkbaren Rechenaufgaben zu lösen. Es würde aber viel zu weit führen, an dieser Stelle auch komplizierte Rechenmethoden zu besprechen, und so wird sich dieses Kapitel im wesentlichen mit den vier Grundrechenarten und den Codewandlungen beschäftigen.

Wirklich vertraut machen müssen Sie sich allerdings schon jetzt mit dem Rechnen mit Zahlen im Zweier-Komplement, da dieses in vielen Befehlen im Mikroprozessor angewandt wird. Im übrigen kann dieses Kapitel immer wieder aufgeschlagen werden, wenn später Programmbeispiele mit Rechnungen oder Codewandlungen besprochen werden.

2.1 Addition und Subtraktion

Grundsätzlich gelten für alle Zahlensysteme, die dem gleichen Bildungsgesetz unterliegen, wie es in Bild 1.5 dargestellt ist, die gleichen Rechenregeln. Am einfachsten kann man dies bei der Addition verfolgen.

Für das Zusammenzählen in dem uns gebräuchlichen *Dezimalsystem* mit den zehn Ziffernsymbolen

0 – 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9

ist das Vorgehen beim schriftlichen Rechnen allgemein bekannt.

```
  742
+ 459
 (111)
------
 1201
```

Man beginnt mit dem Zusammenzählen in der letzten, d. h. der *niederstwertigen Stelle*. Wenn sich die dort ergebende Summe nicht mehr mit den zur Verfügung stehenden zehn Ziffernsymbolen darstellen läßt, erfolgt ein *Übertrag*, englisch CARRY, in die nächsthöhere Stelle, was beim schriftlichen Rechnen durch Einfügen der Übertragszahl in das Rechenschema gekennzeichnet wird. Jetzt werden die Zahlen der zweiten Stelle zusammen mit dem Übertrag addiert, und wenn der Wert wieder höher als neun ist, entsteht ein neuer Übertrag. Auf diese Weise können beliebig große Zahlen miteinander addiert werden, wobei es ohne weiteres auch möglich ist, ganze Zahlenkolonnen zu addieren. Dann kann als Übertrag auch eine höhere Zahl als die Ziffer **1** erscheinen.

In gleicher Weise kann man auch bei allen anderen Zahlensystemen arbeiten. Wenn man das im Abschnitt 1.2 beschriebene *Oktalsystem* betrachtet, so hat dieses insgesamt nur 8 Ziffernsymbole

0 – 1 – 2 – 3 – 4 – 5 – 6 – 7 .

Der Übertrag muß also bereits ausgeführt werden, wenn das Ergebnis in einer Kolonne, die man in diesem Fall als Oktalstelle bezeichnen würde, größer als **7** ist, wie es das folgende Beispiel zeigt.

```
  123
+ 456
  (11)
-----
  601
```

Während man beim Oktalsystem noch leicht nachrechnen kann, ist dies bei Rechnung im *Hexadezimalsystem* schon nicht mehr so einfach möglich, da dieses System mit den 16 Ziffernsymbolen

0 – 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 9 – A – B – C – D – E – F

arbeitet. Der Übertrag erfolgt hier, wenn in einer Kolonne die Summe größer ist, als es der Dezimalzahl **15** gleich hexadezimal **F** entspricht.

```
  1FB
+ 2A4
  (1)
-----
  49F
```

Das oktale und insbesondere das hexadezimale Zahlensystem werden aber in der Computertechnik nicht zum Rechnen benutzt, sondern sie dienen nur als Hilfssysteme zum besseren Verständnis der Binärzahlen für uns Menschen. Es ist daher auch nicht notwendig, sich weiter mit dem Rechnen mit Oktal- oder Hexadezimalzahlen zu befassen, denn moderne Computer rechnen ausschließlich mit *Binärzahlen,* für die aber auch wieder die gleichen Rechenregeln gelten. In diesem System, das oft auch *duales Zahlensystem* genannt wird, gibt es nur die Ziffernsymbole

0 und **1,**

so daß die einzelnen Zahlen sehr viel mehr Stellen haben als in den anderen Zahlensystemen. Der Übertrag muß hier also schon erfolgen, wenn in einer Stelle der dezimale Zahlenwert **2** erreicht ist, wie es das folgende Beispiel zeigt:

```
  100101
+  10111
   (111)
--------
  111100
```

Für das Verständnis der Binärrechnung, wie sie ein Computer durchführt, ist es noch wichtig, zu erkennen, daß ebenso wie beim schriftlichen Rechnen immer mit der am weitesten rechtsstehenden, *niederstwertigen Stelle,* auf englisch LEAST SIGNIFICANT, begonnen wird.

Grundsätzlich kann ein Computer aber nur zwei Zahlen addieren, also keine Kolonnenaddition durchführen, wie wir es in unserem Dezimalsystem bei schriftlicher Addition tun. Wenn ein Computer mehrere Zahlen zu addieren hat, führt er diese Addition nacheinander durch, bildet also zunächst aus der Addition von zwei Zahlen die Zwischensumme und addiert dann zu dieser die nächste Zahl usw. Der Übertrag bei einer Computerrechnung im Binärsystem kann darum auch niemals größer als **1** sein. Er kann also niemals den Informationswert von einem Bit überschreiten.

Im Prinzip gilt das, was über Rechenregeln allgemein gesagt wurde, auch für die *Subtraktion.* Wenn wir im Dezimalsystem schriftlich eine Subtraktion durchführen,

```
  5 68
- 3 82
  (1)
-------
  1 86
```

arbeiten wir auch spaltenweise von rechts beginnend. Es gibt hier jedoch keinen Übertrag, dafür müssen wir *borgen* (englisch BORROW), d.h. von der nächsthöheren Stelle etwas abziehen, wenn der Vorrat an Ziffernsymbolen nicht ausreicht, das Ergebnis einer Stelle auszudrücken.

In genau gleicher Weise kann man auch die Differenz zweier Binärzahlen bilden, d.h. eine Binärzahl von der anderen abziehen.

```
  1 0 0 1 0
- 0 1 1 0 1
 (1)(1)  (1)
-------------
  0 0 1 0 1
```

Wie das Beispiel zeigt, kann in der ersten Stelle ganz rechts von **0** nicht **1** abgezogen werden. Man muß den Wert **1** aus der zweiten Stelle borgen. Dann ergibt sich **10** ($= 2_{10}$) minus **1** gleich **1.** Eine **0** als Ergebnis erscheint in der zweiten Stelle, weil hier von der dort stehenden **1** der vorher geborgte Wert abzuziehen ist. Hierbei braucht nicht weiter geborgt zu

werden, dagegen ist in der dritten und vierten Stelle wieder ein Borgen aus der nächsthöheren Stelle notwendig.

Das eben beschriebene Subtraktionsverfahren ist nun allerdings für Computer nicht besonders geeignet. Die Schaltungsanordnungen, in denen bei Mikroprozessoren arithmetische Operationen durchgeführt werden, sind im allgemeinen so aufgebaut, daß sie einzig und allein nur addieren können. Man verwendet also in der Mikroprozessor-Technik auch für Subtraktionen eine besondere Methode der Addition, die im folgenden Abschnitt beschrieben wird.

2.2 Das Zweier-Komplementsystem

Zunächst einmal ist festzuhalten, daß ein Computer in seiner Rechnerschaltung immer mit *ganzen positiven Zahlen,* die man englisch INTEGER nennt, arbeitet. Es handelt sich dabei immer um reine Binärzahlen mit endlicher Länge, d.h., die Zahl der Binärstellen, die in einem Arbeitsgang verarbeitet werden können, richtet sich nach dem schaltungsmäßigen Aufbau des betreffenden Computers.

Ein Mikroprozessor, in dem eine *Informationsmenge* von 4 Bit parallel verarbeitet wird, kann auch nur eine vierstellige Binärzahl in einer Rechenoperation bewältigen. Der maximale Dezimalwert einer vierstelligen Binärzahl errechnet sich wie folgt:

$$2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15.$$

Wenn man jetzt noch den Wert **0** dazunimmt, können **16** verschiedene Dezimalzahlen, also 2^4, in vier Binärstellen dargestellt werden.

Dementsprechend errechnen sich für:

8 Bit einschließlich 0: 2^8 gleich **256** Dezimalwerte und für 16 Bit einschließlich 0: 2^{16} gleich **65536** Dezimalwerte.

Ebenso wie man sich die in einem Zahlensystem verfügbaren Ziffernsymbole in einem Kreis angeordnet vorstellen kann (Bild 2.1), so kann man auch die Informationen, die in einer bestimmten Anzahl von Binärstellen unterzubringen sind, in einem Kreis anordnen (Bild 2.2). Das bedeutet: ein vierstelliger *Binärzähler* kann höchstens bis zu dem Dezimalwert **15** gleich **1111** zählen. Wenn sein Inhalt bei dieser Zählerstellung noch einmal um den Wert **1** erhöht wird, zeigt er **0000** an.

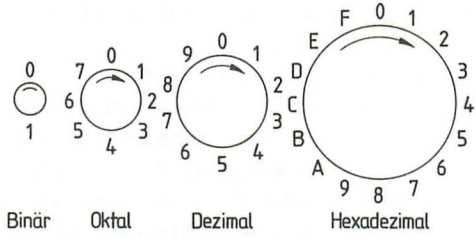


Bild 2.1 Der Ziffernvorrat bei verschiedenen Zahlensystemen
Die verschiedenen in der Computertechnik verwendeten Zahlensysteme unterscheiden sich durch die Menge der jeweils verwendeten Ziffernsymbole. Bei jedem System kann man sich diese im Kreis angeordnet vorstellen. Es wird immer so weit aufwärts gezählt, bis die Zahl der Symbole erschöpft ist, dann muß eine neue Stelle begonnen werden.

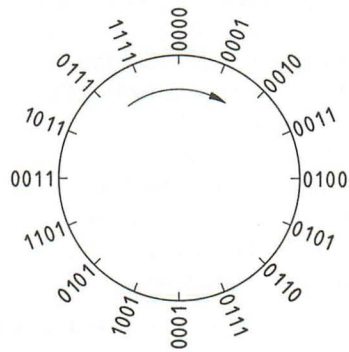


Bild 2.2 Informationsinhalt eine 4-Bit-Binärzählers
In 4-Bit-positionen können 16 verschiedene Informationen dargestellt werden. Beim Aufwärtszählen in einem derartigen, in sich geschlossenen Bitmuster wechselt der Inhalt nach dem höchstmöglichen Zahlenwert **(1111)** wieder zu dem niederstmöglichsten **(0000)** Wert.

Die in der Mikroprozessor-Technik üblichen 8-Bit-Register können also maximal den Dezimalwert **255** enthalten, wenn der Inhalt als ganze positive Zahl interpretiert wird.

Im Grundsatz kann aber nun auch jedes Bitmuster beliebig in anderer Weise interpretiert werden. Man kann z.B., wie es im Bild 2.3 gezeigt wird, die 16 möglichen Bit-Kombinationen eines vierstelligen Bitmusters

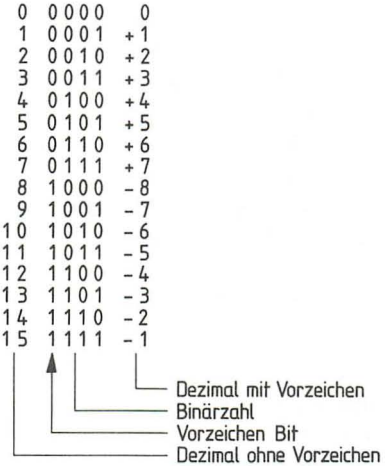


Bild 2.3. Verschiedene Interpretierung einer Vier-Bit-Binärzahl
Die 16 Informationsmöglichkeiten, die eine Vier-Bit-Binärzahl bietet, können in verschiedenster Weise interpretiert werden. So kann man dadurch zum Beispiel die Dezimalzahlen 0 bis 15 ohne Vorzeichen ausdrücken oder aber auch die Zahlen von 0 bis +7 und von −1 bis −8. Bei der im Bild gezeigten Anordnung bedeutet eine **0** in der höchstwertigsten Stelle der Binärzahl, daß damit eine positive Zahl gemeint ist. Eine **1** an dieser Stelle kennzeichnet eine negative Zahl.

so aufteilen, daß sieben davon als positive Zahlen und acht davon als negative Zahlen gelten. In diesem Fall werden nur die drei niederstwertigen Bit für die Zahlendarstellung gebraucht, während das höchstwertige Bit das Vorzeichen symbolisiert.

Auf dem ersten Blick ist nicht einzusehen, was mit dieser Darstellung, die eigentlich eine Beschränkung bedeutet, gewonnen wird. Um die Nützlichkeit dieses Vorgehens zu verstehen, muß man sich etwas mit der *Komplementrechnung* befassen.

In unserem dezimalen Zahlensystem können wir statt

$$7 - 4 = 3$$

auch

$$7 + 6 = (1)3$$

schreiben.

Im zweiten Ausdruck ist die Ziffer **6** der Zahlenwert, der nötig ist, um die Ziffer **4**, die eigentlich subtrahiert werden sollte, zum Zahlenwert der

nächsthöheren Stelle, in diesem Fall **10**, zu ergänzen. Während im ersten Rechenausdruck der Subtrahent **4** wirklich vom Minuenden **7** abgezogen wurde, ist im zweiten Rechenausdruck der Komplementwert des Subtrahenden zum Minuenden addiert worden. Auch in diesem Fall ergibt sich das richtige Ergebnis **3**, wenn man einfach die in der höherwertigen Stelle erscheinende **1** nicht beachtet.

In gleicher Weise kann man z.B. auch bei zweistelligen Zahlen das Komplement zum Wert **100** bilden, dann ergeben sich die folgenden Rechenausdrücke:

85 − 25 = 60 oder **85 + 75 = (1)60**

Ebenso kann man natürlich im Dezimalsystem auch das Komplement zum Wert **1000** oder zu noch höheren Dezimalwerten bilden. Nur würde dies keinem vernünftigen Zweck dienen.

Sehr viel interessanter dagegen ist die Komplementbildung für das Rechnen im Binärsystem, wo im wesentlichen wieder die gleichen Regeln gelten. Statt zu subtrahieren, kann man auch hier mit der richtig gebildeten Komplementzahl addieren. Wenn z.B. die Aufgabe besteht, von der vierstelligen Binärzahl **0101** die Zahl **0011** abzuziehen, so kann man stattdessen die Komplementärzahl zur ersten Binärzahl mit fünf Stellen, also **10000**, bilden und diesen Wert addieren, wie es in dem folgenden Beispiel gezeigt wird:

Binärzahl	**0011** = +3	**0101** = 5
Ergänzung	**1101** = −3	+**1101** = −3
zu **10000**	111	1 1
	10000	(1)0010 +2

Dieses Bit wird nicht berücksichtigt ↑

Wenn jetzt die in dem Beispiel gewählten Zahlen mit der Einteilung in positive und negative Zahlen im Bild 2.3 verglichen werden, erkennt man deutlich, daß die gewählte Einteilung durchaus sinnvoll war, denn der Binärwert der negativen Zahlen entspricht hier dem für die binäre Subtraktion wichtigen *Zweier-Komplement*.

> In der binären Mathematik können Subtraktionen durch Addition der Zweier-Komplementzahl, die sich unter Berücksichtigung der zu verarbeitenden Stellenzahl ergibt, ausgeführt werden.

Das Schöne bei dieser Rechenmethode ist es nun, daß die *Umwandlung einer positiven Binärzahl in ihr Zweier-Komplement,* d.h. in ihren Negativwert, außerordentlich einfach ist. Man braucht hierfür lediglich in der positiven Zahl jeweils die Werte **0** durch Werte **1** und die Werte **1** durch Werte **0** zu ersetzen. Diesen Vorgang nennt man *Komplementieren* oder *Negieren* oder *Invertieren.* Hierdurch wird das sogenannte Einer-Komplement gebildet, zu dem der Wert **1** hinzuzuaddieren ist, wodurch das für die Rechnung maßgebliche *Zweier-Komplement* entsteht, wie es in den folgenden Beispielen dargestellt ist.

Positive Zahl	**0011**	+3	**0101**	+5
Einer-Komplement	**1100**		**1010**	
Addition 1	**1**		**1**	
Negative Zahl	**1101**	−3	**1011**	−5

Grundsätzlich spielt es keine Rolle, mit wieviel Binärstellen gerechnet werden soll. Die *höchstwertige,* d.h. am weitesten links stehende *Stelle,* englisch MOST SIGNIFICANT genannt, dient immer nur zur Kennzeichnung des *Vorzeichens.* Englisch wird diese Position darum oft auch als SIGN-BIT bezeichnet.

> Steht im höchstwertigsten Bit einer Binärzahl eine **1,** so kann sie als negative Zahl im Zweierkomplement interpretiert werden. Eine dort stehende **0** kennzeichnet eine positive Zahl.

Die folgenden Beispiele zeigen dies noch etwas deutlicher.

72 − 31 = 41

+31	0	001 1111
$\overline{31}$	1	110 0000
		1
−31	1	110 0001

+72	0	100 1000
−31	1	110 0001
+41	0	010 1001

↑ **Vorzeichen-Bit** ↑

Das Einer-Komplement wird oft durch einen übergeschriebenen Strich zum Ausdruck gebracht, also:

$\overline{31}$ bedeutet Einer-Komplement von **31.**

Ebenso einfach wie die Umwandlung einer positiven Binärzahl in ihr Zweier-Komplement ist auch die Umwandlung einer Zweier-Komplementzahl in ihren positiven Binärwert. Auch für diesen Wandlungsvorgang ist zunächst das Einer-Komplement zu bilden und dann der Wert **1** hinzuzuaddieren, wie es die folgenden Beispiele zeigen:

Negative Zahl	**1101** −3	**1010** −6
Einer-Komplement	**0010**	**0101**
Addition 1	**1**	**1**
Positive Zahl	**0011** +3	**0110** +6

> Die Bildung des Zweier-Komplements einer Binärzahl erfolgt durch Invertieren aller Bit und anschließende Addition von **1.** Das gleiche Verfahren gilt für die Rückwandlung eines im Zweier-Komplement ausgedrückten Negativ-Wertes in sein positives Äquivalent.

Eine wichtige Sache muß man allerdings beim Rechnen im Zweier-Komplementsystem noch beachten. Die Ergebnisse werden falsch, wenn durch den Rechenvorgang die Zahlenmenge, die in dem betreffenden Bitmuster unterzubringen ist, überschritten wird. Die folgenden Beispiele mögen dies an einigen Rechnungen in einem Vier-Bit-System darstellen.

Beim Beispiel 1 werden die positiven Zahlen **4** und **3** addiert. Das Ergebnis ist **7,** die Kapazität wird noch nicht überschritten, die Rechnung ist gültig. Ein Übertrag findet nicht statt.

	①		②		③
+4	**0100**	+7	**0111**	+7	**0111**
+3	+**0011**	−5	+**1011**	−8	+**1000**
+7	**0111**	+2	(1)**0010**	−1	**1111**

Beim Beispiel 2 wird von dem positiven Wert **7** der Wert **5** subtrahiert. Das Ergebnis ist **+2.** Bei der Subtraktion, die durch Addition des Zweier-Komplements der Zahl **5** ausgeführt wird, entsteht ein Übertrag, was bei der Subtraktion bedeutet, daß nicht geborgt wurde, denn die Zahl **7** ist größer als die Zahl **5.**

Beim Beispiel 3 wird von der Zahl **7** die Zahl **8** abgezogen, auch wieder durch Addition des Zweier-Komplements von **8.** Diesmal entsteht kein Übertrag, was bei der Subtraktion hier bedeutet, daß geborgt werden muß. Da die Zahl **7** kleiner ist als die Zahl **8,** ist ein negatives Ergebnis entstanden, das noch im Zahlenbereich liegt.

④		⑤		⑥	
−4	**1100**	**+5**	**0101**	**−4**	**1100**
−1	**1111**	**+4**	**0100**	**−5**	**1011**
−5	**(1)1011**	**+9**	**1001**	**−9**	**0111**

Beim Beispiel 4 wird von der Zahl **−4** noch der Wert **1** subtrahiert. Beide Zahlen sind als Zweier-Komplement dargestellt und werden addiert. Hierbei entsteht ein Übertrag, was wiederum bedeutet, daß kein Borgen notwendig war, das Ergebnis **−5** liegt im Zahlenbereich.

Beim Beispiel 5 werden wieder zwei positive Zahlen addiert. Das Ergebnis ist **9** und liegt nicht mehr im Zahlenbereich, so daß diese Rechnung unzulässig ist.

Das gleiche gilt für das Beispiel 6. Hier wird von einer negativen Zahl noch einmal eine negative Zahl abgezogen, das Ergebnis **−9** überschreitet den zulässigen Zahlenbereich, so daß die Rechnung auch ungültig ist.

Aus diesen Beispielen sind zwei wichtige Regeln zu erkennen:

> Entsteht bei einer Subtraktion, die durch Addition des Zweier-Komplements ausgeführt wird, kein Übertrag, so bedeutet dies, daß von der nächsthöheren Stelle geborgt werden muß. Ist ein Übertrag entstanden, ist kein Borgen notwendig.

> Das Ergebnis einer Rechnung im Zweier-Komplementsystem ist immer dann falsch, wenn die Vorzeichen Bit der beiden Operanden gleichlautend sind und das Ergebnis ein anderes Vorzeichen hat.

Da die meisten Mikroprozessoren auch Subtraktions-Befehle in ihrem Befehlssatz haben, kann der Programmierer auch bei Subtraktionen den Wert des Subtrahenden als positive Zahl in sein Programm aufnehmen. Die Umwandlung in das Zweier-Komplement erfolgt selbsttätig durch den Mikroprozessor.

Die meisten Mikroprozessoren melden auch durch Setzen des *Überlauf-Bit,* englisch OVERFLOW *(Überfließen),* im Programm-Status-Wort, wenn der erlaubte Zahlenumfang überschritten wird. Hierum braucht man sich aber immer nur bei dem höchstwertigen Zahlen-Byte zu kümmern, denn nur dieses zeigt in seinem höchstwertigen Bit, ob dort eine **0** oder eine **1** steht, ob die Zahl als positiv oder negativ im Zweier-Komplement zu werten ist.

Damit ist die Frage aufgeworfen, wie nun größere Zahlen, die nicht in einem 8 Bit breiten Byte unterzubringen sind, verarbeitet werden können. Da das binäre Zahlensystem ohne weiteres in der Lage ist, auch die größten Zahlen darzustellen, gibt es auch bei einem Mikroprozessor in dieser Hinsicht keine Beschränkung. Man muß dann nur mehrere Byte für die Zahlendarstellung verwenden. Grundsätzlich wird bei der Rechnung immer mit dem niederstwertigen Byte begonnen und ein bei der Rechnung dann entstehender Übertrag bei dem nächstfolgenden Byte mit berücksichtigt, wie es das folgende Beispiel zeigt:

```
  0100 1011     1000 1000
+ 0100 0101     1010 0010
-------------------------
  1001 0001 [1] 0010 1010
```

Das Rechenwerk des Prozessors hält sich dabei ganz genau an die oben aufgeführten Regeln und beachtet, wenn die entsprechenden Rechenbefehle gegeben wurden, dabei die Überträge.

Etwas schwerer zu übersehen sind die Verhältnisse bei Subtraktionsbefehlen, denn hier arbeiten die Mikroprozessortypen etwas unterschiedlich.

Bei einer Subtraktion bedeutet *Borgen,* daß von der nächsthöheren Stelle der Wert **1** abzuziehen ist. Einige Prozessoren setzen darum auch **1** in das Übertragsbit, wenn geborgt werden muß.

Andere Prozessoren beachten die Regel der Zweier-Komplementrechnung, daß geborgt werden muß, wenn kein Übertrag entstanden ist, und es wird dementsprechend eine **0** in das Übertragsbit im Programm-Status-Wort gesetzt, wenn Borgen notwendig ist.

In beiden Fällen wird das Borgen beim nächsten Subtraktionsbefehl immer in der Weise berücksichtigt, daß zu der Zahl, die zu subtrahieren ist, vor der Bildung des Zweier-Komplements der Wert **1** hinzuaddiert wird.

Hierzu das folgende Beispiel:

264	0000 0001	0000 1000
− 16	−0000 0000	0001 0000
248	0000 0000	1111 1000

In einem Mikroprozessor wird diese Rechnung wie folgt durchgeführt:

Bildung des Zweier-Komplements des Subtrahenden im niederstwertigsten Byte	16	0001 0000
	$\overline{16}$	1110 1111
	+	1
	−16	1111 0000
Subtraktion		0000 1000
		+1111 0000
Kein Übertrag, also Borgen		$\boxed{0}$ 1111 1000
Subtrahend im nächsthöheren Byte um 1 erhöhen		0000 0000
		1
		0000 0001
Bildung des Zweier-Komplementes	1	0000 0001
	$\overline{1}$	1111 1110
	+	1
	−1	1111 1111
Subtraktion		0000 0001
		+1111 1111
Übertrag, also kein Borgen		$\boxed{1}$ 1 0000 0000

Wenn mit negativen und positiven Zahlen gerechnet werden soll, muß das Zweier-Komplementsystem Verwendung finden. Um das Vorzei-

chen-Bit beim Ergebnis und um das Überfließen bei einem Rechenvorgang braucht man sich nur zu kümmern, wenn das höchstwertige Byte zur Rechnung ansteht. Das Beispiel im Bild 2.4 möge dies demonstrieren.

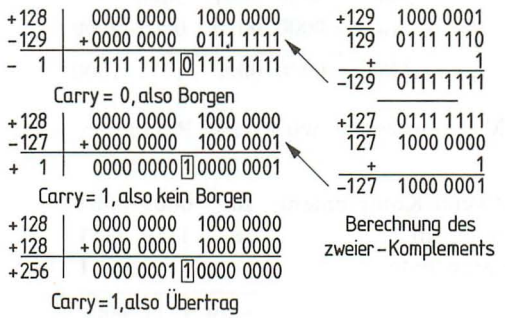


Bild 2.4 Beispiele für Zwei-Byte-Additionen und -Subtraktionen
Die Beispiele zeigen, daß die Ergebnisse auch bei Mehr-Byte-Operationen dann richtig werden, wenn die Regeln für die Zweier-Komplementrechnung nur im niederstwertigsten Byte angewandt werden, weil nur mit kleinen Zahlen gerechnet wird.

Bei dem Beispiel wird mit 16 Bit, also 2 Byte, gerechnet. Man hat für derartige Rechnungen leider einen Ausdruck aus der Computersprache übernommen, der eigentlich in diesem Fall nicht angebracht ist. Man bezeichnet sie als Rechnungen in *doppelter Genauigkeit* (DOUBLE PRECISION).

Der Ordnung halber sei noch darauf hingewiesen, daß es auch Mikroprozessoren gibt, die überhaupt keine Subtraktionsbefehle in ihrem Befehlssatz haben. Selbstverständlich kann man auch mit diesen Prozessoren subtrahieren, nur muß man dann die Umwandlung des Subtrahenden in sein Zweier-Komplement selbst durch entsprechende Befehle programmieren. Man verwendet für diese Mikroprozessoren auch die *Einer-Komplementmethode,* bei der das Einer-Komplement des Minuenden zu bilden ist und der Subtrahend als positive Zahl hinzuaddiert wird. Wie im Bild 2.5 gezeigt, muß nach der Addition durch Abfragen des

Übertrag-Bit entschieden werden, ob die entstandene Zahl als negative Zahl oder als positive Zahl zu werten ist. Wenn kein Übertrag entstanden ist, wird das Ergebnis komplementiert und kann dann als positive Zahl gelesen werden. Ist ein Übertrag entstanden, so ist zum Ergebnis **1** hinzuzuaddieren, und der dann entstandene Binärwert muß als negative Zahl interpretiert werden.

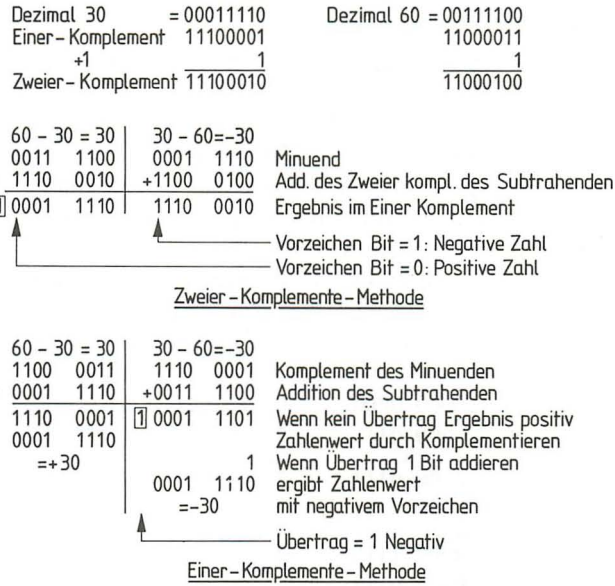


Bild 2.5 Binäre Subtraktionsmethoden
Bei der meistverwendeten Zweier-Komplementmethode erfolgt die Subtraktion durch Addition des Zweier-Komplements. Das Ergbnis ist als Zahl im Zweier-Komplement zu interpretieren.

Demgegenüber muß bei der Einer-Komplementmethode das Ergebnis gegebenenfalls noch korrigiert werden. Es kann als Komplement einer positiven Binärzahl verstanden werden, wenn kein Übertrag entstanden ist. Ein gesetztes Übertragsbit ist ein Zeichen für einen negativen Wert des Ergebnisses, dessen binärer Zahlenwert durch Addition von **1** gefunden wird.

2.3 Multiplikationen und Divisionen

Auch bei diesen beiden Rechenarten sind die Methoden im Dezimal- und Binärsystem gleich. Grundsätzlich kann man jede Multiplikation als wiederholte Addition schreiben.

$$7 \times 3 \text{ entspricht } 3 + 3 + 3 + 3 + 3 + 3 + 3 = 21.$$

Schneller und einfacher ist diese Rechnung allerdings, wenn man die beiden Faktoren vertauscht

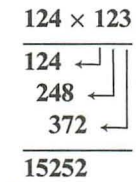
$$3 \times 7 \text{ entspricht } 7 + 7 + 7 = 21.$$

Da ein Computer in seinem Rechenwerk nur addieren kann, ist die Multiplikation durch wiederholte Addition eine Rechenmethode, die auch für die Computertechnik geeignet ist. Der Computer bildet im Binärsystem nach jedem Additionsvorgang die Zwischensumme, zu der dann wieder der zweite Faktor hinzuaddiert wird. Die Zahl der Additionen wird durch einen Zähler, der auf den Wert des ersten Faktors zu setzen ist, kontrolliert, so daß die Multiplikation 3×7 nach dieser Methode im binären Zahlensystem wie folgt aussieht:

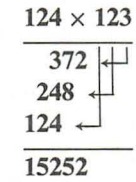
0000 0000	**0**	Zähler = **1**
0000 0111	+ **7**	
0000 0111	**7**	Zähler = **2**
0000 0111	+ **7**	
0000 1110	**14**	Zähler = **3**
0000 0111	+ **7**	
0001 0101	**21**	Ergebnis

Wenn ein derartiges Programm für einen Mikroprozessor geschrieben wird, was bei kleinen Rechnungen durchaus zweckmäßig sein kann, wird man vorher noch untersuchen, welcher der beiden Faktoren kleiner ist. Dieser Faktor kommt dann in den Zähler, während der andere als Summand verwendet wird.

Aber auch bei Multiplikationen gibt es noch eine zweite Rechenmethode, die man bei Dezimalzahlen anwendet. Wenn das Produkt aus den Zahlen **124** und **123** gebildet werden soll, schreiben wir in Deutschland die Zahlen meist wie folgt:

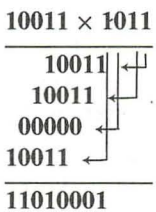


Man multipliziert zunächst den Multiplikanden mit der höchstwertigen Stelle des Multiplikators und schreibt das Zwischenergebnis auf. Danach wird mit den weiteren Stellen des Multiplikators in gleicher Weise multipliziert und dann die Summe der Zwischenergebnisse gebildet. Man kann die gleiche Rechnung nun auch umgekehrt durchführen, indem man mit der niederstwertigen Stelle des Multiplikanden beginnt. Die Rechnung sieht dann wie folgt aus



Wie zu sehen ist, werden hier die Zwischenergebnisse jeweils um eine Stelle nach links verschoben. Das Ergebnis beider Rechnungen ist dann gleich.

Die zweite Methode der Multiplikation wird nun auch in der Computertechnik angewandt. Eine Multiplikation im Binärsystem nach dieser Methode nutzt die Tatsache aus, daß es durch entsprechende Befehle möglich ist, ein gegebenes Bitmuster um eine Position nach links oder rechts zu verschieben.



Bei der *binären Multiplikation* nach dieser Methode beginnt man bei der ersten Teilmultiplikation auch mit dem niederwertigsten Bit des

Multiplikators. Da dieses Bit nur den Wert **1** oder **0** haben kann, ist als Ergebnis nur ein Wert möglich, der entweder dem Multiplikanden entspricht oder aber dem Wert **0.** Wenn nach dieser Methode von einem Computer gerechnet wird, ermittelt er nach jeder Multiplikation sofort die Zwischensumme. Ist der Multiplikand in der betreffenden Stelle **0,** d.h., wenn das Ergebnis der betreffenden Teilmultiplikation auch **0** ist, braucht diese Multiplikation nicht ausgeführt zu werden, sondern es genügt dann, den Multiplikanden eine Stelle nach links zu verschieben. Das Ergebnis entsteht also durch die Kombination von Additions- und Verschiebevorgängen.

Es gibt nun noch ein sehr einfaches weiteres Multiplikationsverfahren, wenn es darum geht, eine Binärzahl nur mit einem anderen kleinen ganzzahligen Wert zu multiplizieren. Die Verschiebung eines Bitmusters, das eine Binärzahl darstellt, um eine Stelle nach links bedeutet nämlich eine *Verdoppelung des Zahlenwerts,* wie es aus folgendem Beispiel zu ersehen ist.

$7 \times 1 = 7$	0000 0111
$7 \times 2 = 14$	0000 1110
$7 \times 4 = 28$	0001 1100
$7 \times 8 = 56$	0011 1000

Wenn man nun z.B. eine gegebene Zahl mit **5** multiplizieren will, so braucht man diese nur um zwei Stellen nach links zu verschieben, was der Multiplikation mit vier entspricht, und danach die Zahl noch einmal hinzuzuaddieren, wie es das folgende Beispiel zeigt.

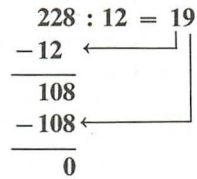
$7 \times 5 = 35$	0001 1100	28
	+0000 0111	+ 7
	0010 0011	35

Auch bei der Division kann man sich wieder an die Methoden, die aus der Dezimalrechnung bekannt sind, halten. Eine Division kann durch wiederholte Subtraktion ausgeführt werden, wobei die Anzahl der durchgeführten Subtraktionen das Ergebnis der Division darstellt.

20 : 5	20 − 5 = 15	1×
	15 − 5 = 10	1×
	10 − 5 = 5	1×
	5 − 5 = 0	1×
		4

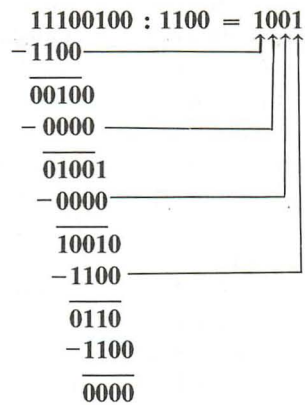
Man kann diese Methode selbstverständlich auch bei Binärzahlen anwenden, nur ist sie nicht gerade besonders elegant, und sie wird darum in der Computertechnik auch nur selten benutzt.

Wie bei der Multiplikation verwendet man darum auch bei der Division meist eine Methode, die auch beim schriftlichen Rechnen mit Dezimalzahlen angewandt wird.



Bei dieser Methode werden zunächst die höchstwertigen Stellen des Dividenden daraufhin geprüft, wie oft der Divisor darin enthalten ist. Die Zahl stellt das Ergebnis dar. Mit ihr ist der Divisor zu multiplizieren, und das Ergebnis dieser Rechnung ist vom Dividenden abzuziehen.

Was nun im Dezimalsystem recht schwierig ist und beim Kopfrechnen einige Mühe macht, ist im Binärsystem sehr viel einfacher, wie das folgende Beispiel einer *binären Division* zeigt.



Auch hier wird zunächst geprüft, ob die höchstwertigen Stellen der Binärzahl es erlauben, den Divisor zu subtrahieren. Auf die beim Dezimalsystem hier noch notwendige nachfolgende Multiplikation kann verzichtet werden, denn das Ergebnis der Multiplikation kann nur den Wert

des Divisors oder den Wert **0** ausmachen. Wenn also eine Subtraktion mit positivem Ergebnis möglich ist, wird als Ergebnis eine **1** eingetragen, der sich evtl. ergebende Rest wird um eine Stelle nach rechts verschoben und der an dieser Stelle des Dividenden stehende Wert hinzuaddiert. Wenn die Subtraktion des Divisors nicht möglich ist, d.h., wenn sich ein negatives Ergebnis ergeben würde, kommt in das Ergebnisfeld eine **0**, und es wird nur verschoben und der Wert der nächstniedrigeren Stelle des Dividenden addiert. Auf diese Weise entsteht durch wiederholtes Subtrahieren und Verschieben das Ergebnis.

Wie bei der Multiplikation, so kann man natürlich auch bei der Division einfache Aufgaben gelegentlich nur durch Verschieben eines Bitmusters durchführen. Möglich ist dies allerdings nur, wenn der Divisor dem Wert der Potenzen der Zahl **2** entspricht.

2.4 Addition und Subtraktion im BCD-Zahlensystem

Bei gewissen einfachen Rechnungen kann man sich die Programmierarbeit erleichtern und grundsätzlich im BCD-Zahlensystem, das in Abschnitt 1.3 schon kurz vorgestellt wurde, arbeiten. Wie die Tabelle in Bild 1.8 zeigt, wird in diesem System jede Dezimalzahl durch eine 4 Bit breite Information dargestellt. In einem Byte können also 2 Dezimalzahlen angeordnet werden. Man spricht dann von *gepackter* (PACKED) *Darstellung*. Für gewisse Rechenverfahren muß man die beiden Halb-Byte dann wieder trennen, was eine *ungepackte Darstellung ergibt.*

Der Vorteil des BCD-Systems liegt darin, daß die Stellenwertigkeit dem Dezimalsystem entspricht, so daß für uns Menschen eine einfache Umcodierung vom Dezimal- in das BCD-System und umgekehrt möglich ist. Dies wird mit dem Nachteil erkauft, daß ein Teil der Informationskapazität, die ein bestimmtes Bitmuster aufweist, verschenkt wird. Darüber hinaus ist für einen Computer das Rechnen mit BCD-Zahlen nicht ganz so einfach, so daß diese Zahlendarstellung im allgemeinen nur für einfache Rechenaufgaben Verwendung findet.

Die Schwierigkeit liegt darin, daß ein Computer in seinen Rechenschaltungen grundsätzlich rein binär arbeitet. Da das BCD-System nur einen Teil der binär möglichen Bit-Kombinationen ausnutzt, müssen bei

jedem Rechenvorgang *Befehle zur Dezimalkorrektur* gegeben werden, die das Ergebnis wieder berichtigen. Man erkennt dies gut bei der Addition.

0101	5
+0111	+7
1100	keine BCD-Zahl
+0110	Dezimalkorrektur
1 0010	12

Wenn die Zahlenwerte **5** und **7** in binärer Form addiert werden, ist das Ergebnis **1100,** was dem Hexadezimalwert **C** entspricht, aber keine BCD-Zahl darstellt. Durch die Addition ist eine *ungültige Tetrade,* also ein Vier-Bit-Muster, das keine BCD-Zahl ist, entstanden. Wenn man jedoch jetzt den Wert **6** als Binärzahl zu dem ungültigen Ergebnis hinzuaddiert, ergibt sich in der letzten Tetrade eine **2** und vorne ein Übertrag, und dieses Bitmuster kann im BCD-System richtig als der Zahlenwert **12** interpretiert werden.

Alle Mikroprozessoren kennen nun Befehle und Schaltungsanordnungen, durch die, wenn nötig, eine Korrektur ausgeführt werden kann.

> Die Dezimalkorrektur bei der BCD-Rechnung ist immer dann notwendig, wenn bei der Addition eine ungültige Tetrade das Ergebnis war, oder wenn ein Übertrag entstanden ist.

Die beiden Halb-Byte werden einzeln nacheinander vom Prozessor überprüft, wenn der entsprechende Befehl zur Dezimalkorrektur programmiert wurde.

Bei allen Mikroprozessoren wird außerdem untersucht, ob ein Halb-Byte-Übertrag, d.h. ein Übertrag aus der Bitposition Nr. 3 in die Bitposition Nr. 4, erfolgt ist. Ist dies der Fall, so wird das entsprechende Bit im Programm-Status-Wort gesetzt und auch eine Dezimalkorrektur durchgeführt, wenn der Befehl dazu im Programm enthalten ist. Die folgenden Beispiel mögen dies noch etwas genauer erläutern.

① 0011 0010 32
+0001 0111 12
0100 1001 49
+0000 0000
0100 1001 49

② 0100 0111 47
+0011 0100 34
0111 1011 7?
+0000 0110
1000 0001 81

③ 1000 1001 89
0010 1000 28
1011 [1] 0001 ?1
0110 0110
[1] 0001 0111 117

Bei dem ersten Beispiel entstehen keine ungültigen Tetraden, so daß keine Dezimalkorrektur notwendig ist.

Beim zweiten Beispiel ist das Ergebnis bei der Addition der vier niederstwertigen Bit größer als **9.** Es entsteht eine ungültige Tetrade, so daß die Dezimalkorrektur durch Addition der Binärzahl **6** erfolgen muß. Da bei der ersten Addition kein Übertrag entstanden ist, und da das Ergebnis auch nach der Korrekturaddition noch kleiner als **9** ist, braucht die erste Tetrade nicht korrigiert zu werden.

Beim dritten Beispiel hat sich bei der Addition der Zahlen in der niederstwertigen Tetrade ein Übertrag ergeben. Dies bedeutet, daß eine Dezimalkorrektur vorzunehmen ist. Da sich nun außerdem auch bei der höherwertigen Tetrade eine Zahl als Ergebnis ergeben hat, die größer als **9** ist, muß auch hier die Dezimalkorrektur durchgeführt werden. Hierdurch ergibt sich dann beim Endergebnis ein Übertrag, da das Ergebnis größer ist als **99,** denn dies ist die größte BCD-Zahl, die noch in einem Byte darstellbar ist.

Wenn ein Prozessor die Dezimalkorrektur bei der BCD-Addition wie oben beschrieben durchführt, ist die Methode für die BCD-Subtraktion etwas komplizierter. Es gibt darum auch Prozessoren, die bei der BCD-Addition ein etwas anderes Verfahren, das im folgenden Beispiel dargestellt wird, verwendet.

0111	**1001**	1. Summand	**79**
+0110	**0110**	Addition 66	
1101	**1111**	Zwischensumme	
+0001	**1000**	2. Summand	**18**
1111	**0111**	Zwischensumme	
+1010	**0000**	Dezimalkorrektur	
1001	**0111**	Summe	**97**

Bei dieser Methode muß zunächst grundsätzlich immer zum ersten Summanden der Hexadezimalwert **66** addiert werden. Danach erfolgt die Addition des zweiten Summanden, gefolgt von dem Befehl zur Dezimalkorrektur. Je nachdem, ob bei der Addition des zweiten Summanden das Carry- und Halbcarry-Bit gesetzt oder gelöscht wurde, addiert der Prozessor nach einem bestimmten Schema* jeweils bei den beiden Halb-Byte noch den Dezimalwert **10** = hexadezimal **A**, wobei in den einzelnen Halb-Byte kein Übertrag berücksichtigt wird. Es entsteht das gleiche richtig zu interpretierende Ergebnis wie bei der anderen Methode.

Der Vorteil der zuletzt geschilderten Korrekturmethode besteht darin, daß Subtraktion im BCD-System sehr einfach zu programmieren sind. Es wird lediglich der normale Subtraktionsvorgang ausgeführt und danach der Befehl zur Dezimalkorrektur gegeben.

[I]		Carry = **1**, da vorher kein «Borgen»	
0100	**0101**	Minuend	**45**
-0011	**1000**	Subtrahend	**38**
[I] **0000**	[0] **1101**	Zwischenergebnis Carry = **1**, kein «Borgen»	
+0000	**1010**	Dezimalkorrektur	
0000	**0111**	Ergebnis	**07**

Etwas schwieriger ist das Vorgehen bei der BCD-Subtraktion, wenn die Prozessoren nach der zuerst geschilderten Methode arbeiten. Das folgende Beispiel zeigt, wie in diesem Fall vorzugehen ist.

---

* Siehe Tabelle für den 2650 im Abschnitt 4.11.

1		Carry = 1, da vorher kein «Borgen»	
	1001 1001	Akku mit **99** Laden	**00**
	+0000 0001	Carry hinzuaddieren	**+ 1**
0	**1001 1010**	Zwischenergebnis	**(9A)**
	−0001 1000	Subtrahenden subtrahieren	**−18**
	1000 0010	Zwischenergebnis	**(82)**
0	**+0111 1001**	Minuenden addieren	**+79**
	1111 1011	Zwischenergebnis	**(FB)**
	+0110 0110	Dezimalkorrektur	**+66**
	0110 0001	Ergebnis	**61**
1			

Wenn ohne Borgen subtrahiert werden soll, muß auch bei dieser Methode zunächst das Carry-Bit auf **1** gesetzt werden. Danach ist in das Register, in dem die Subtraktion ausgeführt werden soll, also im allgemeinen in den Akkumulator, der Hexadezimalwert **99** zu laden. Jetzt wird das Carry-Bit hinzuaddiert und danach der Subtrahend von dem so entstandenen Ergebnis subtrahiert. Im nächsten Arbeitsgang muß dann der Minuend, also die Zahl, von der ursprünglich subtrahiert werden sollte, addiert werden, und wenn dann noch die Dezimalkorrektur ausgeführt wird, ist das Ergebnis richtig. Hierbei muß man berücksichtigen, daß ein gesetztes Carry-Bit in diesem Fall bedeutet, daß nicht geborgt wurde.

2.5 Rechnen mit Dezimalstellen

Bei allen bisherigen Rechnungen wurde unterstellt, daß es sich um ganze Zahlen handelt. In Wirklichkeit muß jedoch sehr oft auch mit *Dezimalbrüchen,* also Zahlen, deren Ziffernfolge durch ein Komma, oder im amerikanischen Sprachgebrauch durch einen Dezimalpunkt, getrennt sind, gearbeitet werden. Grundsätzlich unterscheidet man hierbei die Festkomma- und die Gleit- oder Fließkomma-Rechnung.

Bei *Festkomma-Zahlen* (FIXED POINT) wird vereinbart, daß das Komma immer an der gleichen Stelle der Ziffernreihe steht. Festkomma-Zahlen können darum im wesentlichen wie ganze Zahlen bei der Rechnung behandelt werden. Gegebenenfalls muß die Ziffernfolge vorn bzw.

hinten mit Nullen abgefüllt werden. Bei gewissen Rechenverfahren kann es auch notwendig sein, die Stellung des Kommas noch besonders durch einen Rechengang festzulegen.

Bei der *Gleitkomma-Rechnung* (FLOATING POINT) bedient man sich der *exponentialen Darstellung* für die Zahlenwerte. Hierfür gilt die Formel

$$\mathbf{Z = M \times B^E}$$

in der **Z** die Zahl bedeutet, die durch das Produkt der Mantisse **M** und des Potenzwertes zur Basis **B** des Zahlensystems mit dem Exponenten **E** ausgedrückt wird. Bei dieser Darstellung bestimmt der Wert des Exponenten also die Lage des Kommas, wie die folgenden Beispiele für das Dezimalsystem zeigen.

$$\begin{aligned} 256\,000 &= 0{,}256 \cdot 10^{6} \\ 0{,}256 &= 0{,}256 \cdot 10^{0} \\ 0{,}00256 &= 0{,}256 \cdot 10^{-2} \\ 123{,}456 &= 0{,}123456 \cdot 10^{3} \end{aligned}$$

Man erkennt deutlich, daß die Erhöhung oder Erniedrigung des Exponenten jeweils eine Kommaverschiebung um eine Stelle nach rechts oder links bedeutet.

Die gleiche Zahlendarstellung ist auch für alle anderen Zahlensysteme, die dem gleichen Bildungsgesetz unterworfen sind, möglich, wie die folgenden Beispiele für Binärzahlen zeigen.

$$\begin{aligned} 101{,}1101 &= 0{,}1011101 \cdot 2^{3} \\ 100\,101{,}0 &= 0{,}100101 \cdot 2^{6} \\ 0{,}0001011 &= 0{,}1011 \cdot 2^{-3} \end{aligned}$$

Da das Verschieben eines Bitmusters zu den Operationen gehört, die ein Computer ausführen kann, ist ihm auch das Rechnen in exponentialer Zahlendarstellung möglich. Der Vorteil besteht vor allem darin, daß hierdurch ein sehr viel größerer Zahlenbereich ohne zu großen Speicheraufwand verarbeitet werden kann, als es in der Festkomma-Darstellung möglich ist. Erkaufen muß man dies allerdings mit einem größeren Aufwand für die Programmierung, und so würde es den Umfang dieses Buches sprengen, hierauf im einzelnen einzugehen. Es käme im übrigen auch keinem Programmierer in den Sinn, aufwendige mathematische Programme selbst zu entwickeln. Die entsprechenden

Programme kann man sich vom Hersteller des verwendeten Mikroprozessors beschaffen.

2.6 Codewandlungen

Da der Mikroprozessor intern rein binär arbeitet, wir selbst aber nur im dezimalen Zahlensystem denken und rechnen können, ist es immer wieder notwendig, Binärzahlen in Dezimalzahlen und umgekehrt Dezimalzahlen in Binärzahlen zu wandeln. Es gibt hierfür eine größere Zahl von Methoden, von denen nur einige hier besprochen werden können.

Wenn man ein Programm mit Hilfe eines Entwicklungsgerätes erstellt und durch den in das Gerät eingebauten Computer assemblieren läßt, braucht man sich um die Zahlendarstellung nicht zu kümmern, denn die gesamte Umcodierung wird durch das Assemblierprogramm vorgenommen. Wenn man dagegen ein Programm in der Maschinensprache schreibt, müssen dem Computer auch alle Zahlen in seiner binären Sprache angeboten werden.

Für die Codewandlung kann man am einfachsten Tabellen benutzen, wie sie auch im Anhang dieses Buches abgedruckt sind. Trotzdem ist es oft nützlich, auch Methoden für die Codewandlung zu kennen, um diese gelegentlich bei kleineren Zahlen anwenden zu können.

Wenn man eine Binärzahl in eine Dezimalzahl wandeln will, braucht man nur daran zu denken, daß jeder **1** in einer Binärstelle eine ganz bestimmte dezimale Wertigkeit zuzuordnen ist, die in der folgenden Tabelle für ein 8 Bit breites Bitmuster noch einmal aufgezeigt ist.

7	6	5	4	3	2	1	0	Bit-Nr.
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Wert
128	64	32	16	8	4	2	1	
255	127	63	31	15	7	3	1	Σ-Werte

Im allgemeinen ist es mit einem geringen Rechenaufwand möglich, die Addition dieser Wertigkeiten vorzunehmen, wodurch sich die Dezimal-

zahl ergibt. Bei der Wandlung einer Binärzahl, die viele **1** enthält, kann man auch mit Subtraktionen arbeiten. Wie aus der obenstehenden Tabelle zu ersehen, ist der dezimale Wert einer Binärzahl mit acht **1** gleich **255.** Wenn sich jetzt in dem Bitmuster noch zwei **0** befinden, braucht die Wertigkeit dieser Binärstellen nur von **255** abgezogen werden, um den richtigen Dezimalwert für die ganze Zahl zu erhalten. Beide Methoden sind in den folgenden Beispielen dargestellt.

$$\mathbf{1001\ 0011} = 128 + 16 + 2 + 1 = 147_{10}$$
$$\mathbf{1011\ 1110} = 255 - 64 - 1 \quad = 190_{10}$$

In ähnlicher Weise kann man auch arbeiten, wenn eine Umwandlung von einer Dezimalzahl in eine Binärzahl gewünscht wird. Man subtrahiert hier von der Dezimalzahl nacheinander die Wertigkeiten der einzelnen Binärstellen. Wenn die Subtraktion ausführbar ist, ohne daß ein negativer Rest besteht, kommt in die betreffende Binärstelle eine **1,** ist die Subtraktion nicht ausführbar, wird eine **0** gesetzt. Im Bild 2.6 ist auf der linken Seite diese *Subtraktionsmethode* dargestellt.

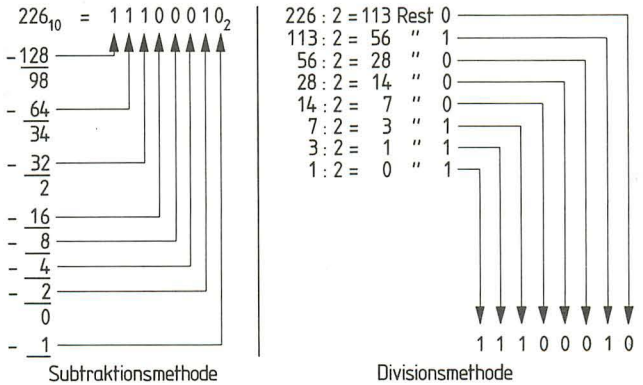


Bild 2.6 Methoden für die Dezimal-Binär-Wandlung
Bei der Subtraktionsmethode wird jeweils ermittelt, ob die Subtraktion eines Prozentwertes der Zahl 2 möglich oder nicht möglich ist. Dementsprechend ist das Binärergebnis **1** oder **0.** Bei der Divisionsmethode wird die Dezimalzahl immer durch 2 dividiert und der Rest notiert. In umgekehrter Reihenfolge gelesen, gibt dieser den Binärwert an.

Für die Codewandlung durch den Mikroprozessor selbst bedient man sich meist einer *Divisionsmethode*. Bei dieser wird die Dezimalzahl laufend durch **2** dividiert. Die Division kann aufgehen, dann entsteht der Rest **0**, oder es kann der Rest **1** entstehen. Diese Reste ergeben die Binärstellen, die so anzuordnen sind, daß der Rest der ersten Division als niederstwertiges Bit der Binärzahl geschrieben wird, wie es auf dem Bild 2.6 rechts gezeigt ist.

3
Aufbau und Wirkung der Befehle für Mikroprozessoren

Die Befehle, die die einzelnen Mikroprozessoren verstehen und ausführen können, sind sehr unterschiedlich, wenn man sie sich in der Maschinensprache ansieht. Aber auch in der *Programmiertechnik* gibt es eine Reihe von *Grundbegriffen,* die für alle Typen gleichermaßen gelten. Der folgende Abschnitt dieses Buches wird diese erklären, ohne daß dabei schon auf die Besonderheiten der einzelnen Prozessortypen eingegangen wird.

Erläutert werden auch die wichtigsten *Fachausdrücke,* die man für das Verständnis *der Software* braucht. Für einige davon gibt es gute und sprachlich zutreffende deutsche Wörter, die dann auch verwendet werden. Andere lassen sich kaum richtig übersetzen, sie müssen als Anglicismen auch im deutschen Text stehenbleiben.

Eine Schwierigkeit besteht nun darin, daß die verschiedenen Hersteller im Englischen ganz verschiedene Ausdrücke verwenden, auch wenn der gleiche Begriff gemeint ist. Das trägt oft zu nicht geringer Verwirrung bei, und so wurde hier versucht, die gebräuchlichen Begriffe nebeneinander zu verwenden, um dadurch das Einprägen der Worte zu erleichtern.

Zwischen den Kapiteln 3 und 4 gibt es gewisse Überschneidungen und Wiederholungen, die durchaus gewollt sind, um durch diese Redundanz das Lernen und Verstehen zu erleichtern.

3.1 Maschinensprache, Assemblersprache und andere Programmiersprachen

Es wurde schon erklärt, daß ein Mikroprozessor nur seine *Maschinensprache* versteht, die für uns Menschen als Binärinformation geschrieben werden kann. Es wurde auch schon gezeigt, daß binäre Informationen ebenso wie Dualzahlen für uns sehr schwer handhabbar sind, und so bedient man sich beim Programmieren, das heißt beim Niederschreiben von Programmen, der sogenannten *Programmiersprachen,* die der menschliche Geist leichter lernen und behalten kann.

Für jeden Mikroprozessor-Typ gibt es nun zunächst eine *Assemblersprache.* In ihr werden die einzelnen Befehle, die der betreffende Prozessor ausführen kann, durch kurze, meist aus 2 bis 4 Buchstaben bestehende Abkürzungen englischer Worte ausgedrückt. Im Englischen bezeichnet man diese *Kürzel* mit dem schwer aussprechbaren, aus der griechischen Sprache stammenden Wort MNEMONICS.

Bild 3.1 zeigt Ausschnitte aus zwei Mikroprozessor-Programmen, die für zwei verschiedene Prozessortypen in den für diese Prozessoren bestimmten Assemblersprachen geschrieben wurden.

Links sieht man zunächst in beiden Beispielen neben der laufenden Zeilennummer die *Adresse,* in der das erste Befehlsbyte abgespeichert ist. Daneben steht dann der Befehl in Maschinensprache, der aus mehreren Byte bestehen kann. Beides ist hexadezimal codiert.

In der Mitte steht der gleiche Befehl in Assemblersprache geschrieben und daneben ein kurzer Kommentar, den jeder Programmierer ganz nach Wunsch hinzufügen kann.

Man sieht, daß trotz gewisser Ähnlichkeiten die Assemblersprachen verschiedener Prozessoren auch ganz verschieden aufgebaut sind. Um die Programmierung in dieser Sprache zu beherrschen, muß man nicht nur die Bedeutung der Kürzel kennen, sondern auch wissen, wie sich die Gesamtdarstellung eines Befehls aufbaut. Man spricht hier auch von der *Syntax,* wobei es gegebenenfalls auf jedes Komma und auf jeden Zwischenraum ankommt.

Jede Befehlszeile in Assemblersprache besteht aus zwei oder drei Teilen, die durch Zwischenräume getrennt sein müssen. Diese drei Teile müssen genau nach den Regeln der betreffenden Sprache geschrieben sein, wenn die Umcodierung in die Maschinensprache des Prozessors durch einen Computer erfolgen soll. Der dann noch folgende Kommen-

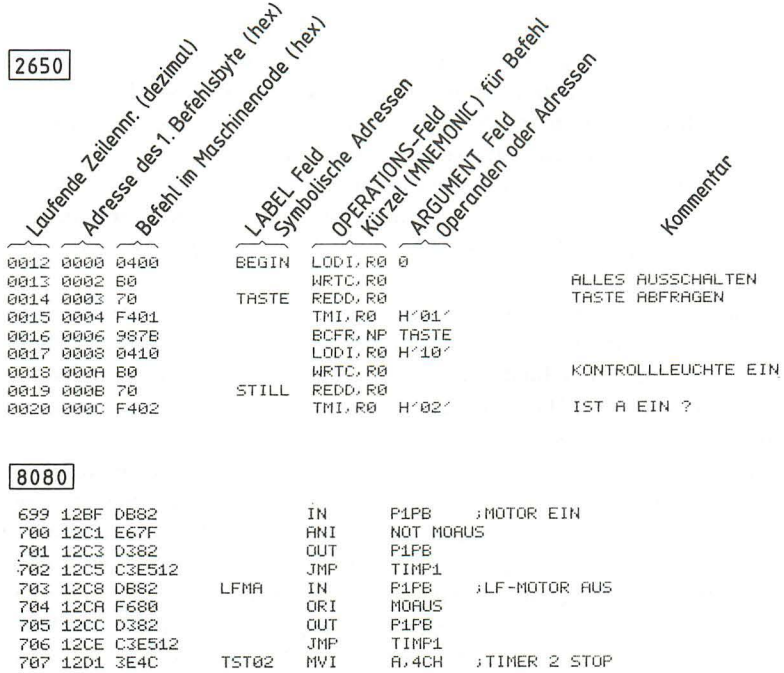


Bild 3.1 Darstellung von Programmen als Assembler-Listing
Die Niederschrift von Programmen in Assemblersprache ist bei den verschiedenen Prozessoren zwar ähnlich, aber doch nicht gleich. Jeder Prozessor hat dabei seine eigene Sprache, und leider werden die gleichen Symbole oft für völlig verschiedene Begriffe verwendet. Wie das Beispiel erkennen läßt, ist die Anordnung der einzelnen Spalten, also das Format der Zeilen bei den verschiedenen Prozessoren, aber fast gleich.

tar wird zwar meistens von dem Computer mitgespeichert und ausgedruckt, hat aber für die Umcodierung keine Bedeutung, so daß er geschrieben werden oder auch weggelassen werden kann.

Zu Beginn des Assemblerausdrucks steht in manchen Zeilen ein Wort, das vom Programmierer beliebig aus Buchstaben und Zahlen gebildet werden kann. In einem Programm darf aber das genau gleiche Wort nicht zweimal vorkommen. Auf Englisch bezeichnet man dieses *Symbolwort,* manchmal auch *Marke* genannt, als LABEL *(Kennzeichen)* oder als NAME *(Name),* was darauf hindeutet, daß der dahinterstehende Befehl nur diesem Symbol zugeordnet ist.

Mit jedem von ihm festgelegten Symbol kann der Programmierer eine ganz bestimmte Adresse kennzeichnen. So ist also das Symbol, das vor

einem Befehl steht, gleichbedeutend mit der Adresse des ersten Befehlsbyte. Wenn eine derartige *symbolische Adresse* einmal festgelegt ist, kann sie an beliebiger Stelle im Programm statt der binär ausgedruckten oder hexadezimal codierten Adresse geschrieben werden.

Nach dem Label folgt im Assemblerausdruck der *mnemonische Kürzel* des Befehls gegebenenfalls zusammen mit der Bezeichnung des im Befehl angezogenen Registers. Man nennt diesen Teil des Assemblerausdrucks auch das *Operationsfeld.*

Nach einem weiteren Zwischenraum kommt dann das *Argumenten-* oder *Operandenfeld.* Es ist in den Assemblersprachen der verschiedenen Hersteller auch verschieden aufgebaut, enthält aber grundsätzlich immer entweder einen *Operanden,* der im Befehl zu verarbeiten ist, oder die Adresse(n), wo der oder die Operanden zu finden sind.

Auch im Operandenfeld verwendet man gern symbolische Adressen, um bestimmte Speicherplätze zu kennzeichnen, oder auch reine Symbole, die dann anstelle von konstanten Zahlenwerten stehen können. Wenn man so arbeitet, ist es allerdings notwendig, alle verwendeten Symbole am Anfang des Programms klar zu definieren. In den meisten Assemblersprachen wird dafür der Ausdruck EQUATE *(gleich),* abgekürzt EQU, verwendet. So bedeutet zum Beispiel MOAUS EQU 80 H, daß überall dort,wo der Programmierer das Wortsymbol MOAUS schreibt, in der Maschinensprache die hexadezimale Zahl 80 eingesetzt werden muß.

Die Erklärung der Assemblersprache als wesentliche Grundlage für das Programmieren von Mikroprozessoren ist die Hauptaufgabe dieses Buches, und so sollen die sogenannten *höheren Programmiersprachen* hier nur kurz erwähnt werden.

Im Gegensatz zur prozessorspezifischen Assemblersprache gibt es auch noch Programmiersprachen, die unabhängig vom Typ des Prozessors, das heißt vom Hardware-Aufbau des Computers, verwendet werden können. Inzwischen gibt es davon schon eine kaum noch übersehbare Anzahl von klangvollen Namen, wie COBOL, BASIC, FORTRAN und viele andere. Sie benötigen immer ein *Übersetzungsprogramm,* das auf einem größeren Computer laufen muß und COMPILER genannt wird. Mit seiner Hilfe erfolgt die Umwandlung des in der Programmiersprache geschriebenen Programms in die Maschinensprache des Computers, auf dem es laufen soll.

Zur Zeit bringt die Verwendung einer höheren Programmiersprache für kleinere und mittlere Mikroprozessor-Systeme kaum Vorteile, da

mehr Speicherplätze benötigt werden und meist langsamer laufende Programme entstehen als bei Programmierung in Assemblersprache.

Grundsätzlich muß die Assemblersprache natürlich auch in die Maschinensprache umcodiert werden. Dies kann mit Bleistift und Papier anhand einer *Befehlstabelle* erfolgen. Bequemer und irrtumsfreier geht dies mit Hilfe eines Computers. Das dafür zu verwendende Programm nennt man *Assembler* oder *Assemblierer.* Mikroprozessor-Programmentwicklungsgeräte arbeiten fast alle in Assemblersprache, die dann mit einem Assemblierprogramm in Maschinensprache umgesetzt wird.

3.2 Befehlssatz und Befehlsschreibung

Ein Mikrocomputer kann ebenso wie jeder andere Computer niemals selbst schöpferisch denken, sondern er kann nur das ausführen, was ihm befohlen wurde, zu tun. Nur der Mensch ist in der Lage, eine Aufgabe zu stellen und sie so für den Computer verständlich zu formulieren, daß er sie lösen kann.

Grundsätzlich kann ein Computer auch nur eine sehr beschränkte Anzahl Operationen ausführen, denn diese müssen bei Entwicklung des Gerätes schon konstruktiv als mögliche Schaltwege vorbereitet sein.

Jede *Operation,* jeder Arbeitsgang eines Computers, muß durch eine entsprechende Anweisung, also einen *Befehl,* englisch INSTRUCTION, ausgelöst werden, und die Folge einer Reihe von Befehlen ergibt dann ein *Programm.* Ziel der Programmierung ist es darum, die unendliche Menge der Aufgaben mit den wenigen zur Verfügung stehenden Ausführungsmöglichkeiten des zu programmierenden Computers in Einklang zu bringen.

Was ein Computer ausführen kann, ist in seinem *Befehlssatz* festgelegt. Er enthält für jeden Befehl den binär verschlüsselten Code, den der Computer versteht, und eine für uns Menschen verständliche Beschreibung, welche Operation ausgeführt wird, wenn der Computer den betreffenden Befehl erhält.

Bei den meisten Mikroprozessoren ergibt sich die Beschränkung auf eine verhältnismäßig kleine Anzahl möglicher Befehle schon allein dadurch, daß auf dem Daten/Befehls-Bus mit 8 Bit Breite nur 256 verschiedene Informationen zu übertragen sind. Aus dem Muster des ersten Befehlsbyte muß der Prozessor aber erkennen, was er zu tun hat.

Die Anzahl der unterschiedlichen *Grundoperationen,* die ein Mikroprozessor ausführen kann, ist meist sogar noch viel kleiner, denn bei allen Prozessor-Typen ist es üblich, daß Befehle in mehreren Variationen ausgeführt werden können. Für jede dieser *Befehlsvarianten* muß aber ein besonderer *Operationscode* zur Verfügung stehen, der den Befehl unzweideutig kennzeichnet.

Obwohl alle Computer im wesentlichen nur die gleichen Grundoperationen ausführen können, sind die Befehlssätze der einzelnen Mikroprozessor-Typen sehr unterschiedlich. Dies liegt einerseits daran, daß der innere Aufbau der Zentraleinheiten der verschiedenen Typen auch sehr verschieden ist und andererseits immer versucht wird, durch geschickte Zusammenfassung mehrerer Operationen in einem Befehl die Programmierung effektiver zu machen.

Bei einigen Prozessor-Typen wird die Zahl der Befehlsvarianten noch durch die Einbeziehung von Programm-Status-Worten in die Befehlsgebung vergrößert. Diese Prozessoren haben ein Programm-Status-Register, das mit beliebigen Bitmustern geladen werden kann. Entsprechend dem Inhalt dieses Registers führt der Prozessor dann gewisse Befehle unterschiedlich aus.

Man muß sich aber davor hüten, die Zahl der möglichen Befehlsvarianten, die ein Prozessor zuläßt, als Maßstab für seine Güte anzusehen. Zu viele Programmiermöglichkeiten wirken manchmal, insbesondere, wenn sie sich nicht aus einer zwingenden Logik ergeben, nur verwirrend. Die meisten Programmierer nutzen sowieso nur einen Teil der möglichen Befehle ihrers Prozessors aus, und ein Anfänger sollte sich darum auch nicht durch die vielen Möglichkeiten, die ein Prozessor bietet, unsicher machen lassen.

Wenn sich die Antwort nicht schon von selbst ergibt, muß jeder Befehl klare Anweisung zu folgenden Fragen enthalten:

Was? Jeder Befehl muß selbstverständlich zum Ausdruck bringen, welche Operation der Prozessor auszuführen hat. Dies ist immer im Operationscode im ersten Byte des Befehls ausgedrückt.

Wer mit Wem? Im Befehl muß klar zum Ausdruck kommen, welche Bitmuster als *Operanden* angesprochen werden. Man versteht hierunter oft auch die Speicherstellen oder Register, in denen die zu verarbeitenden Daten als Bitmuster stehen. Es gibt Befehle, bei denen diese Daten

direkt im Befehl stehen, oft ergibt sich die Operanden-Adresse aber auch erst durch einen Rechenvorgang oder aus dem Inhalt eines bestimmten Registers.

Wohin? Grundsätzlich muß festgelegt werden, wo das *Ergebnis einer Operation* abgespeichert werden soll. Bei einigen Befehlen liegt dies durch die Befehlsdefinition bereits fest, bei anderen muß die Zieladresse im Befehl genannt werden.

Nächster Befehl? Normalerweise holt der Prozessor den nächsten Befehl aus der nächstfolgenden Speicheradresse, weil der dafür zuständige Befehlsadreß-Zähler nach Übernahme eines Befehlsbyte selbsttätig immer um 1 erhöht wird. Es gibt jedoch Befehle, bei denen das Programm in anderer Weise weiterlaufen soll. In diesem Fall muß die *Adresse, von der aus das Programm weitergehen soll,* aus dem Befehl zu ersehen sein.

Bedingung erfüllt? Entscheidend für die Arbeitweise eines Computers ist es, daß er in der Lage ist, zu prüfen, ob eine vorgegebene Bedingung erfüllt ist oder nicht. Je nach der Entscheidung kann dann das Programm in verschiedener Weise weiterlaufen. Gegebenenfalls muß also im Befehl zum Ausdruck gebracht werden, *welche Bedingung zu prüfen ist* und was bei erfüllter bzw. nichterfüllter Bedingung zu geschehen hat.

3.3 Befehlsformat und Befehlsaufbau

Bei den meisten 8-Bit-Mikroprozessoren gibt es im Befehlssatz *Befehle verschiedener Länge.* Sie können 1 Byte, 2 Byte oder 3 Byte lang sein und beanspruchen dementsprechend auch 1, 2, oder 3 Byte Speicherraum. Kurze Befehle werden darum beim Programmieren auch bevorzugt angewandt, wenn dies möglich ist.

Der Aufbau der Befehle wird durch das für jeden Prozessor vorgeschriebene *Befehlsformat* bestimmt (Bild 3.2). Die Länge der Befehle ergibt sich dabei wiederum aus der Informationsmenge, die der Prozessor für die Ausführung des Befehls benötigt. Sie wird durch die Art der *Adressierung* bestimmt. Man versteht hierunter die Angabe der Stelle, wo der Operand oder die Operanden zu finden sind.

Grundsätzlich enthält das erste Byte eines Befehls immer den *Operationscode,* aus dem der Prozessor entnimmt, was er zu tun hat. Bei der *Registeradressierung,* bei der sich alle Vorgänge innerhalb der CPU

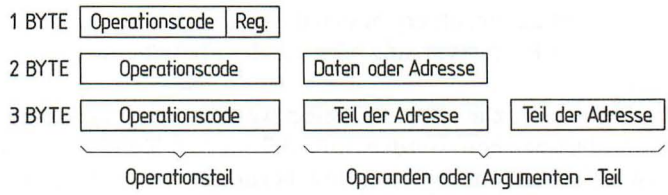


Bild 3.2 Befehlsaufbau (Befehlsformate)
Der Aufbau der Befehle ist bei den einzelnen Prozessortypen unterschiedlich. Grundsätzlich enthält das erste Befehlsbyte aber immer den Operationscode, aus dem der Prozessor erkennt, was er zu tun hat. Soweit notwendig, muß dieser Teil auch die beteiligten Register spezifizieren. Das zweite und das dritte Byte dienen der Aufnahme des Operanden oder der Operandenadresse.

abspielen, enthält das erste Byte auch Angaben darüber, welche Register an der Operation beteiligt sind. Man spricht hier von dem *Quellregister,* aus dem die Daten für die Operation kommen, und dem *Zielregister,* in das sie gehen.

Einfach zu verstehen ist auch die *unmittelbare Adressierung.* Bei einem *2-Byte-Befehl* mit dieser Adressierungsart stehen die zu verarbeitenden oder zu transportierenden Daten unmittelbar im zweiten Byte des Befehls. Das erste Befehls-Byte enthält dann neben dem Befehlscode gegebenenfalls noch in codierter Form das Datenziel oder die Adresse des zweiten Operanden für eine Datenverarbeitung.

Wenn im Befehl die *volle Operandenadresse* angegeben werden muß, werden *3 Byte* benötigt, denn bei den meisten Mikroprozessoren steht ein Adreßraum mit 16 Bit Adressiermöglichkeit zur Verfügung. Auch bei diesen Befehlen muß gegebenenfalls die Datenquelle bzw. das Datenziel oder der zweite Operand aus dem Operationscode zu entnehmen sein.

Es gibt noch weitere Befehlsformate für andere Adressierarten, die später besprochen werden.

3.4 Die vier wichtigsten Befehlsarten

Der weitaus größte Teil aller Befehle, die Mikroprozessoren verarbeiten können, läßt sich in die folgenden Gruppen von Befehlsarten eingliedern:

- Datentransport-Befehle
- Datenverarbeitungs-Befehle
- Prüf- und Vergleichs-Befehle
- Programmablauf-Befehle

3.4.1 Datentransport-Befehle

Einen wesentlichen Anteil in jedem Mikrocomputer-Programm machen die *Datentransport-* oder, wie man auch sagt, die *Datentransfer-Befehle* aus. Bei ihnen werden als Bitmuster dargestellte Daten ohne jede Veränderung nur von einem Platz zu einem anderen transportiert. Dies ist

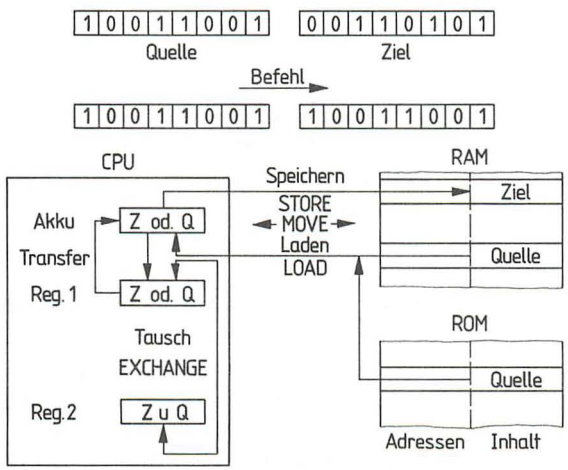


Bild 3.3 Datentransport
Für alle Datentransport-Operationen gibt es eine Datenquelle und ein Datenziel. Durch die Operation wird der Inhalt der Quelle in das Ziel transportiert. Es gibt sowohl Befehle für den Datentransport innerhalb der CPU-Register als auch für den Datentransport zwischen den Speichern und der CPU.

notwendig, da die eigentliche Datenverarbeitung, bei der eine Änderung des Bitmusters erfolgt, nur in der CPU vorgenommen werden kann, so daß ein ständiger Datentransfer zwischen dem Prozessor und den *ROM- und RAM-Speichern,* zusammenfassend im Englischen MEMORY genannt, notwendig ist.

Im allgemeinen bezeichnet man den Datentransport von der CPU zu den Speichern mit STORE *(Speichern)* und den Transport in der Gegenrichtung mit LOAD *(Laden),* jedoch wird auch das Wort MOVE (bewegen) für beide Transportrichtungen verwendet (Bild 3.3).

Bei jedem Datentransport gibt es eine *Datenquelle* und ein *Datenziel,* worunter man die Speicherplätze oder Register versteht, aus der die Daten kommen, bzw. in die sie durch den Befehl transportiert werden sollen.

Während im Amerikanischen die Datenquelle einheitlich auch als SOURCE bezeichnet wird, kennt man dort für das Datenziel die Ausdrücke DESTINATION *(Bestimmung, Ziel)* und DRAIN *(Abfluß, Senke),* so daß diese Worte auch in manchen deutschen Veröffentlichungen erscheinen.

> Grundsätzlich ist zu merken, daß bei jedem Datentransport der alte Inhalt des Datenziels verlorengeht und durch das Bitmuster der Datenquelle ersetzt wird. Datenquelle und Datenziel haben nach dem Befehl den gleichen Inhalt.

Bei den meisten Prozessoren können Daten auch zwischen den einzelnen Registern der CPU hin- und hertransportiert werden. Wenn der Transport in den Akkumulator erfolgt, spricht man auch hier von *Laden* und in umgekehrter Richtung von *Speichern.* Es werden aber auch die Ausdrücke TRANSFER und MOVE verwendet.

Eine besondere Art von Transportbefehlen sind die bei einigen Prozessoren möglichen *Tauschbefehle* (EXCHANGE INSTRUCTIONS). Bei ihnen findet ein echter Austausch der Inhalte statt. Abweichend von der allgemeinen Regel übernimmt hier das Quellenregister den Inhalt des Zielregisters und das Zielregister den Inhalt des Quellenregisters. In diesem Fall sind also nach dem Befehl die Inhalte auch wieder ungleich.

Einige Prozessoren kennen besondere Kurzbefehle, um den Inhalt von Registern in einen dafür reservierten Speicherbereich, den *Stapelspeicher,* englisch STACK, abzuspeichern bzw. wieder dort auszulesen. Auf diese Befehle wird später noch näher eingegangen.

Oft wird auch in einem Befehl Datenverarbeitung mit einem Transportvorgang verbunden. Datentransport von einem ROM in ein RAM oder innerhalb eines RAM sind dagegen nur bei ganz wenigen Prozessoren mit einem Befehl möglich.

Eine Abart der Datentransportbefehle stellen die *Ein-/Ausgabe-Befehle* dar, die auf englisch INPUT/OUTPUT INSTRUCTIONS genannt werden. Im Prinzip entsprechen sie den anderen Datentransport-Befehlen, nur werden dabei durch den Prozessor die Steuerleitungen etwas anders geschaltet. Im Englischen werden diese Befehle auch mit WRITE für *Schreiben* = *Ausgeben* und READ für *Lesen* = *Eingeben* bezeichnet.

3.4.2 Datenverarbeitungs-Befehle

Im Gegensatz zu den reinen Transportbefehlen bewirken alle *Datenverarbeitungs-Befehle eine Veränderung der als Daten zu interpretierenden Bitmuster nach ganz bestimmten Regeln.* Bei den meisten Befehlen dieser Gruppe werden zwei Bitmuster miteinander verknüpft, und es entsteht als Ergebnis ein drittes Bitmuster. Es gibt aber auch Datenverarbeitungs-Befehle, bei denen nur ein Datenbyte beteiligt ist.

Bei allen Datenverarbeitungs-Befehlen, die eine Verknüpfung zum Ziel haben, gibt es zwei Operanden, die der Prozessor verarbeiten muß (Bild 3.4). Im Befehl müßten darum eigentlich auch die Adressen der beiden Operanden angezogen werden, wie dies auch oft bei größeren Computern üblich ist. Die meisten Mikroprozessoren sind aber in ihrer Struktur so aufgebaut, daß bei der Datenverarbeitung ein Operand immer im Akkumulator oder in einem anderen Speicher mit akkumulatorähnlichem Aufbau enthalten sein muß. So spricht man bei diesen Befehlen meist nur von dem Operanden und der Operanden-Adresse und meint damit eigentlich den 2. Operanden.

Die eigentliche Verknüpfung der beiden Bitmuster geschieht in der ALU der *Arithmetik-Logik-Einheit,* einer Kombination von elektronischen Bauelementen in der CPU. Immer enthalten die Datenverarbeitungs-Befehle auch gewisse Transportoperationen, da der Inhalt des

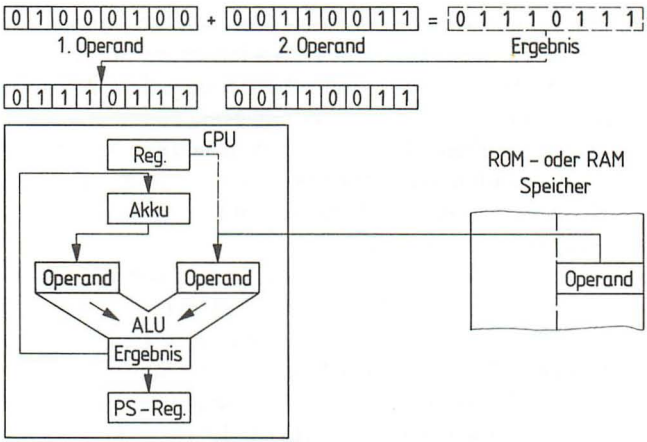


Bild 3.4 Datenverarbeitung
Bei den wichtigsten Datenverarbeitungs-Befehlen sind jeweils zwei Operanden beteiligt, deren Bitmuster nach bestimmten Regeln miteinander verknüpft wird. Grundsätzlich wird das Ergebnis dieser Verknüpfung, das wieder ein Bitmuster ist, dort abgespeichert, wo sich vorher der erste Operand befand. Bei den meisten Mikroprozessoren ist dies der Akkumulator. Die eigentliche Datenverarbeitung findet in der Arithmetik-Logik-Einheit (ALU) statt. Gewisse Ergebnisse der Datenverarbeitung werden in das Programm-Status-Register gemeldet.

Akkumulators und der im Befehl genannte Operand in die ALU transportiert werden müssen.

> Kennzeichnend für alle Datenverarbeitungs-Befehle, durch die zwei Bitmuster verknüpft werden, ist es, daß das Ergebnis der Verknüpfung immer dort zu finden ist, wo der erste Operand stand. Im allgemeinen ist dies der Akkumulator, dessen alter Inhalt verlorengeht. Dagegen bleibt der Inhalt des im Befehl angezogenen 2. Operanden enthalten.

Das Ergebnis der Datenverarbeitungs-Operation wird aber nicht nur im Akkumulator abgespeichert, sondern es wird auch noch nach ganz bestimmten Merkmalen beurteilt. So stellt der Prozessor z.B. fest, ob bei

der Operation im mathematischen Sinn ein Übertrag erfolgt ist. Je nach dem Ergebnis dieser Prüfung wird dann im *Programm-Status-Register,* oft auch FLAG REGISTER genannt, ein ganz bestimmtes Bit auf **1** oder **0** gesetzt.

Nun gibt es noch zwei Arten von Datenverarbeitungs-Befehlen, bei denen immer nur der Inhalt eines Registers oder einer Speicherstelle beteiligt ist. Soweit es sich um die Veränderung des Inhalts eines Registers handelt, sind es meist kurze 1-Byte-Befehle, bei denen das gewünschte Register durch den Operationscode bestimmt wird.

In diese Gruppe gehören zunächst die Befehle, bei denen der Binärwert des betreffenden Bitmusters jeweils um den Wert **1** erhöht bzw. erniedrigt wird. Man nennt dies *Incrementieren (Erhöhen)* bzw. *Decrementieren (Erniedrigen).*

Einige Prozessoren erlauben es auch, durch einen Befehl ein Bitmuster zu *komplementieren.* Man versteht darunter die Vertauschung der Werte in allen Bitpositionen. Aus allen **0** wird also eine **1,** und alle **1** werden zu **0.**

In einigen Befehlssätzen finden sich auch Befehle, mit denen der Inhalt eines Registers oder einer Speicherstelle auf Null gesetzt werden kann. Auf Englisch bezeichnet man diese *Nullstell-Befehle* als CLEAR INSTRUCTIONS.

Die zweite Gruppe von Datenverarbeitungs-Befehlen, bei denen nur ein Operand beteiligt ist, sind die *Rotations-* und die *Schiebe-*(SHIFT-) *Befehle.* Bei ihnen wird das Bitmuster durch den Befehl jeweils in einer ganz bestimmten Weise zyklisch verschoben, wie im Abschnitt 3.6.3 im einzelnen erklärt wird.

3.4.3 Prüf- und Vergleichs-Befehle

Im Aufbau den Datenverarbeitungs-Befehlen mit zwei Operanden ähnlich sind die *Prüf- und Vergleichsbefehle,* in ihrer Wirkung sind sie aber unterschiedlich (Bild 3.5). Auch hier werden immer zwei Bitmuster miteinander verknüpft, es wird aber nicht ein Ergebnis gespeichert, sondern nur die entsprechenden Bit im Programm-Status-Register auf **0** oder **1** gesetzt. Oft verwendet man für Vergleichsbefehle als Operanden ein im Befehl selbst stehendes Bitmuster, das dann *Maske* (MASC) genannt wird.

In ähnlicher Weise arbeiten auch gewisse *Test-Befehle* mit einer Mas-

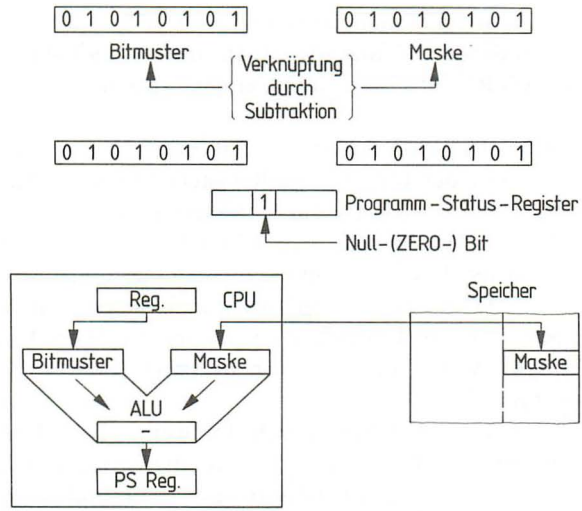


Bild 3.5 Vergleichsoperationen
Der Vergleich zweier Bitmuster erfolgt im allgemeinen durch Subtraktionsverknüpfungen. Die beiden Bitmuster bleiben erhalten. Das Ergebnis wird durch Setzen oder Löschen eines Bit im Programm-Status-Register angezeigt.

ke. Sinn dieser Befehle ist es, innerhalb eines Bitmusters ganz bestimmte Bitpositionen dahingehend zu prüfen, ob sie den Inhalt **0** oder **1** haben.

> Kennzeichnend für alle Test- und Vergleichsbefehle ist es, daß die Bitmuster, die geprüft oder verglichen werden, durch den Befehl nicht verändert werden. Das Ergebnis des Vergleichs oder der Prüfung wird jeweils im Programm-Status-Register angezeigt.

3.4.4 Programmablauf-Befehle

Eine völlig andere Wirkung als die bisher besprochenen Befehle haben *Programmablauf-Befehle,* die auch mit *Sprungbefehl* oder *Verzweigungsbefehl,* englisch JUMP oder BRANCH, bezeichnet werden. Durch sie

wird keinerlei Veränderung in den Arbeitsregistern oder Datenspeichern bewirkt. Beeinflußt wird nur der Inhalt des *Befehlsadreß-Zählers,* des internen Registers also, das immer die Adresse des als nächstes zu verarbeitenden Befehls enthält.

Statt der Adresse eines Operanden, eines Datenziels oder einer Datenquelle enthalten Sprungbefehle in den dem Operationscode folgenden Byte die Adresse, die in das Befehlsadreß-Register zu laden ist.

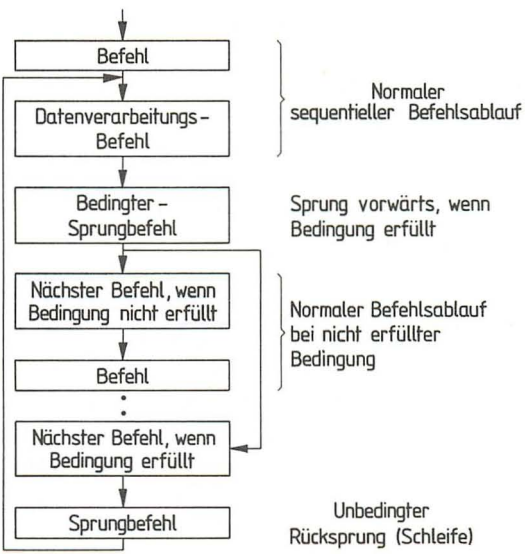


Bild 3.6 Sprungbefehle

Während der Prozessor normalerweise einen Befehl nach dem anderen, wie sie in aufsteigender Adreßreihenfolge im Programmspeicher stehen, ausführt, wird er durch Sprungbefehle veranlaßt, an einer anderen Stelle des Adreßraums weiter zu arbeiten. Es gibt bedingte Sprungbefehle, bei denen der Sprung erfolgen darf, wenn die durch den Programmierer gestellte Bedingung erfüllt ist, und unbedingte Sprungbefehle. Grundsätzlich sind sowohl Sprünge vorwärts, das heißt in Richtung höherwertiger Adresse, als auch rückwärts möglich.

Diese Adresse bestimmt den Speicherplatz, aus dem sich der Prozessor den nächsten zu verarbeitenden Befehl holt, wodurch der Befehl, der sonst in normaler sequentieller Folge an der Reihe wäre, übersprungen wird (Bild 3.6).

Sprungbefehle können sowohl *Vorwärtssprünge,* also in Richtung einer höherwertigen Adresse, veranlassen als auch *Rücksprünge* auf eine niedrigere Adresse bewirken. Je nach Prozessortyp sind auch bei ihnen verschiedene Adressierungsarten möglich, auf die noch im einzelnen eingegangen wird.

In jedem Befehlssatz gibt es *unbedingte und bedingte Sprünge.* Bei *unbedingten Sprungbefehlen* muß der Sprung in jedem Fall ausgeführt werden. Bei diesen wird der normale, sequentielle Programmablauf nur an einer anderen Stelle des Programmspeicherbereichs fortgesetzt.

Sehr wichtig sind die *bedingten Sprünge,* bei denen der Prozessor selbst entscheidet, ob es das Programm in normaler Weise mit dem nächstfolgenden Befehl fortführt, oder ob zu einem ganz anderen Programmteil gesprungen wird. Grundsätzlich kann es sich bei einem einzelnen Sprungbefehl immer nur um eine *JA/NEIN-Entscheidung* handeln. Das heißt also, der Prozessor prüft vor Ausführung des Sprungbefehls, ob die im Befehl angezogene Bedingung erfüllt oder nicht erfüllt ist, was englisch meist mit TRUE und FALSE und im Deutschen mit «*JA*» und «*NEIN*» bezeichnet wird.

Die Prüfung der Sprungbedingung erfolgt im allgemeinen durch Prüfen einer im Programm-Status-Register vorher gesetzten Binärinformation. Durch unterschiedliche Operationscodes kann dann zum Beispiel entweder ein Programmsprung veranlaßt werden, wenn die Bedingung erfüllt ist, oder aber auch, wenn die Bedingung nicht erfüllt wurde.

Das Setzen der Binärinformation, also das Einschreiben einer **1** oder einer **0** in eine ganz bestimmte Bit-Position des Programm-Status-Registers, erfolgt entweder selbsttätig durch den Prozessor nach jeder Datenverarbeitungs-Operation oder durch einen Prüfbefehl.

Sehr ähnlich wie Sprungbefehle sind die Befehle für den *Anruf eines Unterprogramms* aufgebaut. Auch bei diesen kann bedingt oder unbedingt in einen anderen Programmteil gesprungen werden. Sie werden englisch oft mit CALL bezeichnet. Der Hauptunterschied gegenüber normalen Sprungbefehlen besteht darin, daß *beim Anruf eines Unterprogramms die normalerweise im Programm nächstfolgende Befehlsadresse gespeichert* wird. Durch einen besonderen *Rücksprungbefehl,* englisch

RETURN genannt, kann der Prozessor veranlaßt werden, die gespeicherte Befehlsadresse wieder in das Befehlsadreß-Register zu laden und das Programm an der durch das Unterprogramm unterbrochenen Stelle fortzusetzen (siehe Abschnitt 3.7).

3.5 Bedeutung des Programm-Status-Wortes

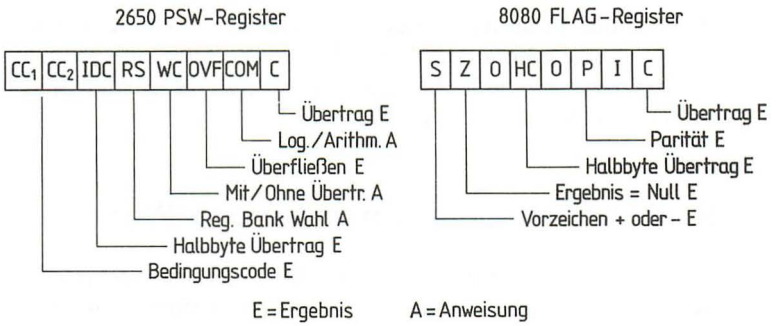
Jeder Mikroprozessor hat ein *Register für das Programm-Status-Wort (PSW).* Wie der Name schon sagt, wird in ihm eine Information abgespeichert, die den augenblicklichen Status des Prozessors zum Ausdruck bringt. Das Register enthält also bestimmte Aussagen über die Arbeit des Prozessors. Im Prinzip hat jedes Bit im PSW eine bestimmte Bedeutung, jedoch werden bei einigen Prozessoren nicht alle möglichen Bitpositionen ausgenutzt (Bild 3.6).

Während einige Prozessortypen im PSW-Register nur die *Ergebnisse* der zuletzt ausgeführten Operation anzeigen, indem sie in bestimmte Bitpositionen eine **0** oder **1** setzen, verwenden andere Typen das Programm-Status-Wort auch, um dem Prozessor gewisse *Anweisungen* für seine weitere Arbeit zu geben. In diesem Fall kann also der Prozessor die Befehle in verschiedener *Arbeitsweise,* englisch MODE, abarbeiten, je nachdem, ob in einer bestimmten Bitposition eine **0** oder **1** gesetzt wurde.

Bei den Prozessoren, die mit Anweisungen im PSW arbeiten, besteht auch immer eine Möglichkeit, durch Befehle den Inhalt des Programm-Status-Registers zu beeinflussen. Man spricht dann oft von *Setzen* (SET), wenn eine Bitposition mit **1** geladen wird, und von *Löschen* (CLEAR), wenn eine **0** eingeschrieben wird.

Bei allen Prozessoren gibt es die folgenden Ergebnis-Aussagen im Programm-Status-Wort (Bild 3.7):

- *Übertrag* – CARRY. Ein Bit wird gesetzt, wenn sich bei einer Addition ein Übertrag ergeben hat, oder wenn bei einer Subtraktion **nicht** geborgt wurde. Das Übertrags-Bit wird auch durch gewisse Rotations- und Schiebebefehle beinflußt.
- *Halbbyte-Übertrag* – HALF-CARRY, INTERMEDIATE oder INTERDIGIT-CARRY. Ein Bit wird gesetzt, wenn bei Rechenoperationen ein Übertrag vom Bit Nr. 3 auf das Bit Nr. 4 notwendig



3.7 Aufbau von Programm-Status-Registern
Die Programm-Status-Register der verschiedenen Prozessoren sind unterschiedlich. In einigen, wie z.B. beim *8080*, werden nur die Ergebnisse von Datenverarbeitungs-Operationen angezeigt. Bei anderen wiederum, wie z.B. beim *2650*, dient das Programm-Status-Register auch dazu, dem Prozessor bestimmte Anweisung für die Befehlsausführung zu geben, je nachdem, ob in bestimmten Positionen eine **1** oder eine **0** eingeschrieben wird.

geworden ist. Diese Anzeige wird für Rechenoperationen im BCD-System benötigt.

- *Null* – ZERO. Durch Setzen eines Bit wird angezeigt, daß bei der letzten Operation als Ergebnis der Wert **00** entstanden ist.
- *Vorzeichen* – SIGN. Beim Rechnen mit Zahlen im Zweier-Komplement bestimmt das Bit Nr. 7 das Vorzeichen. Negative Zahlen werden durch eine **1** in dieser Position des PSW gekennzeichnet.

Einige Prozessoren kennen noch weitere Ergebnisaussagen. So kann z.B. *gerade* (EVEN) oder *ungerade* (ODD) *Parität* beim Ergebnis der letzten Operation gemeldet werden; was bedeutet, daß die Zahl aller **1** im Bitmuster gerade bzw. ungerade ist. Andere Prozessoren setzen ein Bit, wenn ein OVERFLOW *(Überfließen)* vorgekommen ist. Man versteht darunter, daß beim Rechnen im Zweier-Komplement ein ungültiges Ergebnis entstanden ist (siehe Abschnitt 2.2).

Bezüglich der im Programm-Status-Wort möglichen Anweisungen unterscheiden sich die einzelnen Prozessortypen sehr erheblich, aber hierauf wird später noch im einzelnen eingegangen. Wenn Anweisungen im Programm-Status-Wort gegeben werden können, erhöht sich die Zahl der möglichen Befehlsvarianten beträchtlich. Auf der anderen Seite muß man dann auch beim Programmieren ganz besonders auf das PSW achten.

3.6 Möglichkeiten der Datenverarbeitung

Ein Mikroprozessor kann eigentlich nur ganz wenige Grundbefehle, in denen Daten verarbeitet werden, ausführen. Dies ist dadurch bedingt, daß nur Operationen, die durch elektrische Schaltzustände darstellbar sind, als Befehle ausgeführt werden können. Im wesentlichen gibt es die drei folgenden Möglichkeiten der Datenverarbeitung:

- *Binäre Addition*
- *Logische Verknüpfungen*
- *Verschieben eines Bitmusters*

In den Befehlssätzen der Mikroprozessoren findet man meist noch einige weitere Datenverarbeitungs-Befehle, die aus mehreren der oben genannten Grundoperationen zusammengesetzt sind. Sie werden dann immer durch ein kleines im Mikroprozessor-Baustein selbst gespeichertes Ablaufprogramm gesteuert, um das der Programmierer sich im einzelnen nicht zu kümmern braucht.

So haben die meisten Prozessoren bereits *Subtraktionsbefehle* und einige neue Prozessoren sogar *Multiplikations-* und *Divisionsbefehle* in ihrem Befehlssatz. Bei diesen Befehlen werden immer eine Reihe von Datenverarbeitungs-Operationen nacheinander ausgeführt, ohne daß dies im einzelnen nach außen hin sichtbar wird.

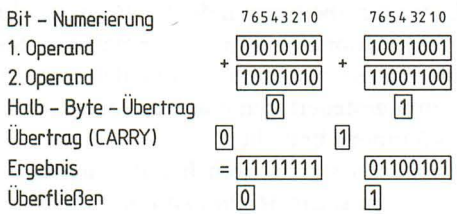
Wenn es die genannten Datenverarbeitungs-Befehle im Befehlssatz eines Mikroprozessors nicht gibt, ist es Aufgabe des Programmierers, durch Kombination der zur Verfügung stehenden Befehle den Prozessor zu veranlassen, auch Datenverarbeitungs-Operationen durchzuführen, die nicht von vornherein in seinem Befehlssatz vorhanden sind. Schon an dieser Stelle kann gesagt werden, daß es mit den wenigen genannten Grundoperationen möglich ist, praktisch alle in der Mathematik bekannten Rechenmethoden durchzuführen oder sonstige Veränderungen von Daten in beliebiger Weise vorzunehmen.

3.6.1 Addition und Subtraktion im Zweier-Komplement

Ein Mikroprozessor arbeitet grundsätzlich nur mit binären Informationen. Ohne weitere Codewandlung können diese auch als Zahlendarstellung im dualen Zahlensystem, das heißt also als Binärzahlen, verstanden werden. Da der Prozessor als mathematische Grundoperation nur Additionen ausführen kann, werden bei einem 8-Bit-Prozessor jeweils zwei

Binärzahlen mit acht Stellen durch einen Additionsbefehl zusammengezählt. Je nachdem, ob dabei ein Übertrag entsteht, ist das Ergebnis eine acht- oder eine neunstellige Binärzahl.

Die Additionselemente eines Mikroprozessors, die in der *Arithmetik-Logik-Einheit* enthalten sind, erkennen einen Übertrag und setzen dann an einer bestimmten Stelle im Programm-Status-Wort eine **1.** Ebenso wird grundsätzlich dort eine **0** eingeschrieben, wenn kein Übertrag bei der Addition entstanden ist. Man spricht hier im allgemeinen vom *Setzen* und *Löschen* des *Carry-Bit* (Bild 3.8).



3.8 Beeinflussung der Bedingungsbit bei einer Additionsoperation
Bei den meisten Prozessoren werden durch arithmetische Operationen drei Bedingungsbit im Programm-Status-Wort beeinflußt. Das Übertragsbit wird bei Additionen gesetzt, wenn ein Übertrag aus dem Bit Nr. 7 in das nächsthöhere Bit notwendig wird. Ebenso wird ein Bit in der Anzeige für den Halb-Byte-Übertrag gesetzt, wenn ein Übertrag aus dem Bit Nr. 3 in das Bit Nr. 4 erfolgt ist. Überfließen wird durch Setzen einer **1** gemeldet, wenn beim Rechnen im Zweier-Komplement ein ungültiges Ergebnis entstanden ist.

Bei allen Additionsoperationen wird vom Prozessor aber auch noch festgestellt, ob ein Übertrag aus der Bitposition Nummer 3 in die Bitposition Nr. 4 notwendig geworden ist. Man spricht hier vom *Halbbyte-Übertrag* (HALFBYTE CARRY). Im Programm-Status-Wort gibt es dann auch eine Position, in der ein Halbbyte-Übertrag durch Setzen einer **1** gekennzeichnet wird. Ebenso wird dort die **1** wieder zur **0** rückgesetzt, wenn an dieser Stelle kein Übertrag entstanden ist.

Als drittes prüfen einige Prozessoren noch, ob durch die Rechenoperation, wenn das Bitmuster als Zahl im Zweier-Komplement zu interpretieren ist, ein unzulässiges Ergebnis entstanden ist. Unzulässig und damit falsch ist das Ergebnis einer Addition immer dann, wenn die Vorzeichen Bit der beiden zu addierenden Zahlen den gleichen Wert

haben, also **1** und **1** oder **0** und **0** lauten, und dann beim Ergebnis ein entgegengesetzter Wert entstanden ist. Man bezeichnet diesen Vorgang als *Überfließen* (OVERFLOW).

Es gibt auch noch Prozessoren, bei denen der Wert des Bit Nr. 7, also des Vorzeichen-Bit, besonders ausgewertet wird. In diesem Fall besitzt das Programm-Status-Wort noch eine Bitposition, die im allgemeinen mit *Negativ* bezeichnet wird. Da eine **1** im Bit Nr. 7 bei Zweier-Komplementdarstellung einen negativen Wert bedeutet, wird im Programm-Status-Wort dort jeweils eine **1** gesetzt, wenn das Ergebnis der Operation eine negative Zahl ergeben hat, und dementsprechend erscheint dort eine **0**, wenn das Ergebnis positiv war.

Die Mikroprozessoren, die in ihrem Befehlssatz auch *Subtraktions-Befehle* kennen, führen diese im allgemeinen durch Addition des Zweier-Komplements der zu subtrahierenden Zahl aus. Die Umwandlung des Subtrahenden, also der zu subtrahierenden Zahl in ihr Zweier-Komplement wird vom Prozessor selbst durchgeführt. Der Programmierer gibt bei Subtraktionsbefehlen auch den Subtrahenden als positive Zahl an. Der Minuend, also die Zahl, von der subtrahiert wird, muß immer mit ihrem wahren Wert, d.h., wenn es sich um eine positive Zahl handelt, mit einer **0** als Vorzeichen-Bit, und bei negativer Zahl mit einer **1** als Vorzeichen-Bit, in das Programm eingegeben werden, wenn im Zweier-Komplement gearbeitet wird.

> Bei der Subtraktion wird das Übertragsbit verschieden behandelt. Einige Prozessoren, wie der *2650* und der *6502*, setzen es auf **1**, wenn kein Übertrag von der nächsthöheren Stelle, also kein Borgen, erforderlich war. Bei anderen Prozessoren wiederum wird das Übertragsbit immer dann auf **1** gesetzt, wenn geborgt werden mußte. Nach der letzgenannten Regel arbeiten z.B. der *6800* und der *8080*.

Bei einer reinen Addition ist die Rechenkapazität eines 8-Bit-Prozessors auf dezimal **255** begrenzt. Wird mit Vorzeichen gerechnet, so kann in 8 Bit als äußerstes die Dezimalzahl **+127** bzw. **−128** dargestellt werden. Mit entsprechenden Befehlen kann man jedoch die im Carry-Bit gespeicherte Information mit in den Rechenvorgang übernehmen und dann

auch mit beliebig großen Zahlen, die sich dann eben über mehrere Byte erstrecken, rechnen. Hierauf wird im Kapitel 5 noch näher eingegangen.

3.6.2 Logische Operationen

Die sogenannten *logischen Verknüpfungen* sind das eigentliche Arbeitsgebiet des Hardware-Schaltungsentwicklers, aber auch ein Mikroprozessor kann eine Reihe derartiger Verknüpfungen durchführen. Der Logikzustand **1** kann bedeuten, daß ein Strom fließt, d.h. ein Schalter eingeschaltet ist, und der Logikzustand **0**, daß kein Strom fließt, also ein Schalter ausgeschaltet wurde. Man kann so durch zwei oder mehr Schalter, die entweder in Serie oder parallel angeordnet werden, die verschiedensten logischen Verknüpfungen darstellen, und der Mikroprozessor arbeitet in der Tat dementsprechend, nur werden keine einfachen Schalter, sondern schnell schaltende Transistoren verwendet.

Es gibt in den Befehlssätzen der Mikroprozessoren gewöhnlich drei logische Verknüpfungen, die durch Befehle veranlaßt werden können. Hierbei sind jeweils zwei Bitmuster beteiligt, wobei man das zweite oft als *Maske* bezeichnet. Grundsätzlich wird das Bit Nr. 0 des einen Bitmusters mit dem Bit Nr. 0 des anderen Bitmusters, Bit Nr. 1 wieder mit Bit Nr. 1 usw. verknüpft.

Bei jeder Verknüpfung entsteht wieder ein Ergebnis, das bei der *UND-Verknüpfung,* englisch AND, nur dann eine **1** zeigt, wenn beide Bitmuster auch eine **1** zeigen.

Demgegenüber erscheint bei der *ODER-Verknüpfung,* englisch OR, im Ergebnis eine **1,** wenn eines oder beide Bitmuster eine **1** an der betreffenden Stelle ausweisen.

Als drittes gibt es dann noch die *NUR-ODER-Verknüpfung,* englisch EXCLUSIV OR, bei der nur dann als Ergebnis eine **1** erscheint, wenn eines der beiden Bitmuster eine **1** und das andere eine **0** hat (Bild 3.9).

Etwas verwirrend ist es, daß als Zeichen für diese logischen Verknüpfungen leider Symbole aus verschiedenen Systemen verwendet werden. Am häufigsten verwendet werden die nach oben und unten weisenden Winkel, wie sie in Bild 3.9 dargestellt sind. Um diese Zeichen besser zu behalten, kann man daran denken, daß der nach oben zeigende Winkel (∧) dem Buchstaben A ähnelt und somit als Symbol für die im Engli-

zu verbinden		UND AND ∧	ODER OR ∨	EXCLUSIV ODER EXR ⊻
Inhalt	Maske	Ergebnis		
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0
		1010 1010 ∧0101 1010 0000 1010	1010 1010 ∨0101 1010 1111 1010	1010 1010 0101 1010 1111 0000
		0 in der Maske setzt Ergebnis auf **0**; bei **1** bleibt Inhalt stehen	**1** in der Maske setzt Ergebnis immer auf **1**; bei **0** bleibt Inhalt stehen	**1** in der Maske komplementiert bei **0** bleibt der Inhalt stehen

Bild 3.9 Logische Verknüpfungen
Jeder Mikroprozessor kann zwei Bitmuster, von denen das zweite oft Maske genannt wird, nach den Regeln der logischen Verknüpfungen miteinander verbinden. Im Bild sind die dafür anzuwendenden Regeln und einige Beispiele dargestellt.

schen AND genannte UND-Verknüpfung verwendet wird. Der nach unten weisende Winkel (V) kennzeichnet dann die ODER- und der gleiche Winkel mit einem Querstrich die NUR-ODER-Verknüpfung (⊻).

Sehr unglücklich ist das zweite Symbol-System für diese Verknüpfungen. Hier wird die UND-Verknüpfung durch einen Punkt ·, die ODER-Verknüpfung durch ein Kreuz + und die NUR-ODER-Verknüpfung durch Kreuz in einem Kreis ⊕ dargestellt. Noch unglücklicher sind die Schaltzeichen für logische Verknüpfungselemente, die nach einem neuen Normblatt für Schaltpläne verwendet werden sollen. Da diese Zeichen aber bei der Mikroprozessor-Programmierung keine Rolle spielen, soll auf sie nicht weiter eingegangen werden.

Mit Hilfe der logischen Verknüpfungen lassen sich Bitmuster verändern, wie auch im Bild 3.9 dargestellt ist. Man benötigt diese Art der Datenverarbeitung insbesondere dann, wenn jedes der Bit in einem Bitmuster als Einzelinformation eine gewisse Bedeutung hat. Man kann so zum Beispiel mit Hilfe der UND-*Verknüpfung einzelne Bit ausblenden,* d.h. auf **0** setzen. Überall, wo in der Maske eine **0** steht, erscheint im Ergebnis

auch eine **0**, auch wenn in dem Bitmuster, dessen Inhalt verändert werden sollte, eine **1** stand.

Dementsprechend kann man durch die ODER-*Verknüpfung einem Bitmuster eine* **1** *aufzwingen*, die überall dort erscheint, wo die Maske eine **1** hat.

Die *Exklusiv-ODER-Verknüpfung* kann verwendet werden, um ein ganzes Bitmuster oder einzelne Bit darin zu *komplementieren*, d.h. jeweils aus einer **0** eine **1** bzw. aus einer **1** eine **0** zu machen.

3.6.3 Rotieren und Schieben

Die dritte Art der Datenverarbeitung, die in der Mikroprozessor-Technik verwendet wird, besteht im zyklischen Verschieben eines Bitmusters nach gewissen Regeln, wie es in Bild 3.10 dargestellt ist. Man unterscheidet zwischen *Rotieren* (ROTATE) und *Verschieben* (SHIFT). Während alle Mikroprozessoren die Rotationsbefehle kennen, sind die Verschiebebefehle nicht in allen Befehlssätzen enthalten.

Bei der Rotation wird der Inhalt eines Registers gelegentlich auf einer Speicherstelle durch einen Befehl immer um eine Position entweder nach links oder nach rechts verschoben, wobei je nach Befehl der Inhalt des Carry-Bit mit eingeschlossen oder nicht eingeschlossen sein kann. Wenn die Verschiebung nicht durch das Carry-Bit geht, ist nach acht gleichen Rotationsbefehlen der ursprüngliche Inhalt wiederhergestellt. Bei Rotationen mit Carry werden hierzu neun Befehle benötigt.

Auch bei Schiebebefehlen wird das Bitmuster durch einen Befehl um eine Position verschoben. Der Unterschied zu den Rotationsbefehlen besteht darin, daß das Bitmuster am Ende durch Nullen aufgefüllt wird, so daß nach acht bzw. neun Verschiebungen nur noch Nullen im Bitmuster stehen. Bei einigen Prozessoren gibt es noch Varianten dieser Schiebebefehle, die für gewisse mathematische Operationen zweckmäßig sind. Man unterscheidet dabei zwischen logischem und arithmetischem Verschieben.

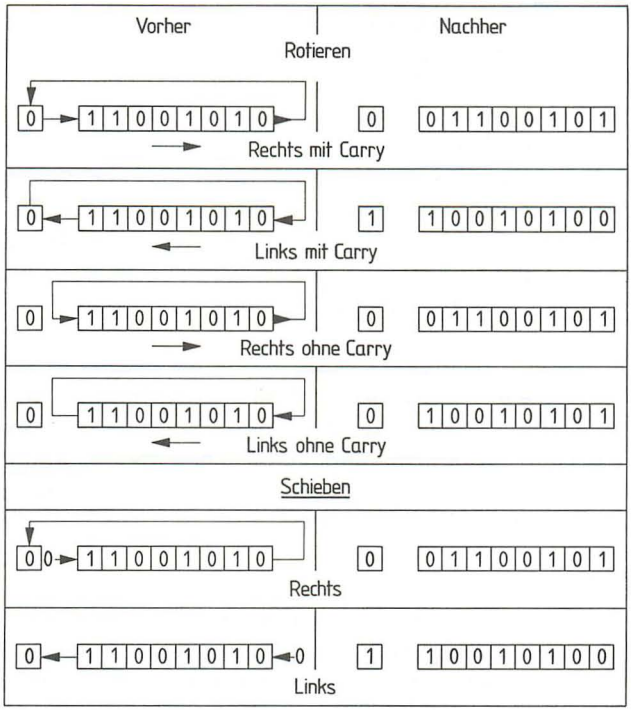


Bild 3.10 Rotieren und Schieben
Jeder Mikroprozessor kennt eine Reihe von Befehlen, durch die ein zyklisches Verschieben eines Bitmusters bewirkt wird. Das Bild zeigt einige typische Rotations- und Verschiebeoperationen. Während die Information beim Rotieren quasi in einem Kreisprozeß umläuft, wird beim Schieben ein Bit nach dem anderen herausgeschoben und das Bitmuster hinten durch Nullen aufgefüllt. Es gibt aber noch andere Schiebeoperationen, als sie hier dargestellt wurden.

3.7 Unterprogramm-Anruf und Stapelspeicher-Bedienung

Jeder Mikroprozessor verarbeitet Befehle, mit denen ein *Unterprogramm,* englisch SUBROUTINE, angerufen werden kann. Man versteht darunter einen in sich abgeschlossenen Programmteil, der zur Erfüllung der gestellten Aufgabe mehrfach benötigt wird.

Es ist offensichtlich, daß durch die Unterprogrammtechnik nicht nur Programmierarbeit, sondern auch Speicherplätze gespart werden. Denn in jedem etwas größeren Mikroprozessor-Programm wird es Programmabschnitte geben, die mehrfach verwendet werden können. Man denke hierbei z.B. an Rechenfunktionen, aber auch an die Programmteile, die zum Auslesen einer Tastatur oder zur Anzeige notwendig sind.

Im Prinzip ist der Anruf eines Unterprogramms einem Sprungbefehl vergleichbar. Im Befehl muß die Adresse stehen, die als nächste Befehlsadresse in den Befehlsadreß-Speicher zu übernehmen ist, damit der Prozessor dort den nächsten zu verarbeitenden Befehl findet.

> Der Unterschied zwischen einem normalen Sprungbefehl und einem Unterprogramm-Anruf besteht darin, daß der Prozessor nach Beendigung des Unterprogramms seine Arbeit dort fortsetzt, wo das Hauptprogramm durch den Sprung in das Unterprogramm unterbrochen wurde.

Um das Hauptprogramm in dieser Weise fortsetzen zu können, muß der Prozessor die Befehlsadresse, die normalerweise nach dem *Einsprungbefehl* in das Unterprogramm an der Reihe gewesen wäre, speichern, um sie nach der Beendigung des Unterprogramms wieder in den Befehlsadreßzähler laden zu können. Die Speicherung dieser *Rücksprungadresse* geschieht bei allen Prozessoren durch einen besonders organisierten Speicher, der *Stapelspeicher,* manchmal auch *Kellerspeicher,* englisch STACK, genannt wird. Dieser Speicher arbeitet nach dem Prinzip LIFO (LAST IN FIRST OUT), was – *zuletzt ein, zuerst aus* – bedeutet. Es handelt sich also um einen Speicher, bei dem die zuletzt abgespeicherte Information zuerst wieder ausgegeben wird.

Erreicht wird dies dadurch, daß man mit einem *Stapelzeiger* (STACK POINTER) arbeitet. Man versteht darunter ein Register, das jeweils die Adresse einer Speicherstelle enthält, in die eine Information abgespeichert werden soll oder aus der eine Information zu entnehmen ist.

Es gibt Prozessoren, bei denen der Stapelspeicher, der die Rücksprungadressen aufnehmen kann, auf den Prozessorchip integriert ist. Bei anderen Prozessoren wird jeweils ein bestimmter Bereich in einem RAM als Stapelspeicher verwendet. *Unterprogramme* lassen sich auch *ineinander*

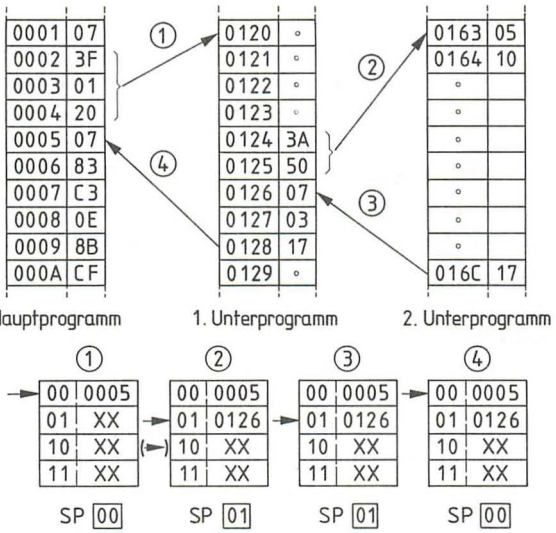


Bild 3.11 Anruf von Unterprogrammen und Funktion des Stapelspeichers
Die meisten Mikroprozessoren erlauben es, einzelne Programmteile wiederholt als Unterprogramm anzurufen und sogar mehrere Unterprogramme ineinander zu verschachteln. Im Bild wird die Funktion des Stapelspeichers und des Stapelzeigers (SP), die beim *2650* auf dem Prozessorchip integriert sind, dargestellt.
Durch den Befehl ① im Hauptprogramm wird das erste Unterprogramm mit seinem ersten Befehl in der Adresse **0120** angerufen. Gleichzeitig wird die Adresse des nächstfolgenden Befehls, also **0005,** an der Stelle des Stapelspeichers abgelegt, auf die der Stapelzeiger zeigt. Danach wird der Stapelzeiger automatisch um den Wert **1** erhöht und zeigt jetzt auf den Speicherplatz **01.**
Im ersten Unterprogramm wird durch den Befehl ② das zweite Unterprogramm angerufen. Wieder wird die im Programm folgende Adresse, diesmal **0126,** im Stapelspeicher abgespeichert und der Stapelzeiger wieder um **1** erhöht, so daß er jetzt auf den nächsten freien Stapelplatz zeigt.
Im zweiten Unterprogramm befindet sich jetzt der Rücksprungbefehl ③, der zunächst bewirkt, daß der Stapelzeiger um den Wert **1** erniedrigt wird und jetzt wieder auf die Adresse **0126** zeigt, an der das erste Unterprogramm fortgesetzt wird.
Wenn jetzt auch dieses Programm zum Rücksprungbefehl kommt ④, wird der Stapelzeiger wieder um 1 erniedrigt und zeigt jetzt auf die Adresse **0005,** mit der das Hauptprogramm dann weiterläuft.

verschachteln. Auf englisch spricht man dann von NESTED SUBROUTINES, wie es im Bild 3.11 dargestellt ist.

Mit Hilfe eines entsprechenden Befehls (1) wird zunächst das erste Unterprogramm angerufen. In diesem Befehl gibt es dann wieder einen Befehl (2), mit dem ein weiteres Unterprogramm angerufen wird. Der

Stapelzeiger, der im allgemeinen entsprechend seiner englischen Bezeichnung mit SP abgekürzt wird, enthält vor dem ersten Unterprogramm-Anruf die Adresse **00,** so daß in dieser die dem Unterprogramm-Anruf-Befehl folgende Adresse abgespeichert wird. Danach wird der Stapelzeiger um **1** erhöht und zeigt somit auf die Adresse **01.** Wenn jetzt ein weiterer Unterprogramm-Anruf erfolgt, wird in dieser Adresse die nächste Rücksprungadresse niedergelegt.

Wenn das zweite Unterprogramm beendet ist, muß dieses durch einen Rücksprungbefehl abgeschlossen werden. Dieser Befehl bewirkt, daß die Adresse, auf die der Stapelzeiger zeigt, in den Befehlsadreß-Speicher übernommen wird und daß das Programm also von dieser Adresse aus weiterläuft (3). Nach der Adreßübernahme wird der Inhalt des Stapelzeigers um den Wert **1** vermindert und zeigt nunmehr auf die nächste Rücksprungadresse, an der das Hauptprogramm fortgesetzt wird, wenn das erste Unterprogramm durch den Rücksprungbefehl als beendet gekennzeichnet ist (4).

Bei dem hier geschilderten Verfahren wird der Inhalt des Stapelspeichers jeweils nach dem Abspeichern einer Adresse erhöht und vor dem Auslesen der Adresse erniedrigt. Es gibt auch andere Verfahren, den Stapelspeicher zu bedienen, die im Kapitel 4 noch näher erläutert werden.

3.8 Die verschiedenen Adressierarten

Die Ausführung und Wirkung der einzelnen Befehle wird weitgehend durch die *Adressierarten,* englisch ADDRESSING MODE, bestimmt. *Man versteht hierunter die Art, in der im Befehl festgelegt ist, wie und wo der Prozessor den oder die Operanden findet.*

Bei dem Operanden kann es sich auch um eine Registeradresse oder um eine Speicheradresse handeln, in der dann erst der eigentliche Operand, d.h. die zu transportierende oder zu verarbeitende Information, steht. Sogar reine Sprungadressen werden gelegentlich als Operanden bezeichnet, während es eigentlich nur eine Adressierart gibt, bei der der Operand selbst unmittelbar im Befehl steht.

Bei Befehlen, die zwei Operanden benötigen, wird die Adresse des einen oft schon durch den Befehlscode ausgedrückt, wobei es sich dann immer um ein Register handelt. Es gibt nun Assemblersprachen, bei

denen dieses Register auch im Operationsfeld steht (siehe hierzu auch Bild 3.1). Bei anderen Assemblersprachen wiederum enthält das Operationsfeld nur den mnemonischen Kürzel, während alle Angaben über die Operanden auch im Operandenfeld stehen. Die verschiedenen Mikroprozessor-Typen unterscheiden sich etwas bezüglich der Adressierarten, die bei ihnen zulässig sind. Leider werden auch insbesondere in der englischsprachigen Literatur sehr verschiedene Ausdrücke für die gleiche Adressierart verwendet. Im folgenden wird die Wirkung der einzelnen Adressierarten zunächst im Prinzip erklärt, ohne daß auf die Besonderheiten der einzelnen Prozessortypen eingegangen wird. Eine genaue Beschreibung der bei den verschiedenen Prozessoren gegebenen Adressiermöglichkeiten folgt dann in Kapitel 4.

3.8.1 Registeradressierung

Jeder Prozessor kennt kurze Ein-Byte-Befehle, die in diesem einen Byte alle für die Befehlsausführung notwendigen Angaben enthalten. Im Englischen spricht darum auch von IMPLIED oder INHERENT ADDRESSING, was mit *«Im Befehl enthalten»* übersetzt werden kann.

Bei Datentransport-Befehlen (Bild 3.12) enthält der Operationscode dann in verschlüsselter Form sowohl das Ziel- als auch Quellregister. Im Speicher wird nur eine Speicherstelle benötigt, die den gesamten Befehl aufnimmt. Nach Ausführung des Befehls haben beide Register den gleichen Inhalt.

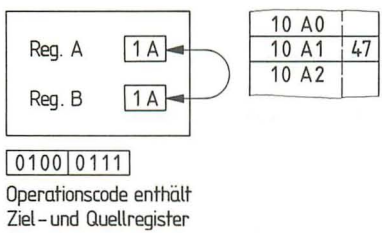


Bild 3.12 Register-Adressierung
Befehle mit Register-Adressierung sind Ein-Byte-Befehle. Alle Angaben über Ziel- und Quellenregister sind im Operationscode enthalten.

Ebenso gibt es auch Datenverarbeitungs-Befehle mit Registeradressierung. In diesem Fall werden die Inhalte zweier Register miteinander verknüpft, wobei meist der Akkumulator als eines der Register beteiligt

ist. Im allgemeinen dient der Akkumulator dann auch als Ergebnisregister.

Bei Datenverarbeitungs-Befehlen, die nur einen Operanden haben, also z.B. bei allen Rotations- und Schiebebefehlen, aber auch bei den Increment- und Decrement-Befehlen, bei denen der Inhalt eines Registers jeweils um den Wert 1 erhöht oder erniedrigt wird, genügt meist auch ein Befehlsbyte, um dem Prozessor alle notwendigen Informationen für den Befehl zu übermitteln.

3.8.2 Adressierung durch Registerpaar

Einige Mikroprozessoren erlauben es, zwei interne Register zu einem *Registerpaar* zusammenzufassen, so daß dort dann eine 16 Byte breite Information unterzubringen ist. Diese Information reicht aus, um eine Adresse in dem üblichen System eines 8-Bit-Prozessors genau darzustellen. Es gibt darum bei diesen Prozessortypen eine Art indirekte Adressierung, bei der der Operand für den Befehl in der Adresse zu finden ist, die in dem betreffenden Registerpaar steht (Bild 3.13). *Das Registerpaar dient also als Zeiger, der auf die Operandenadresse zeigt.*

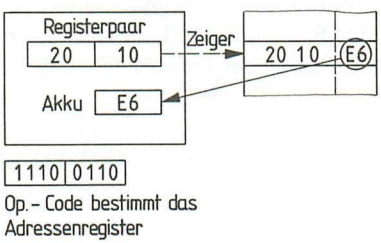


Bild 3.13 Adressierung durch Registerpaar
Durch ein Registerpaar kann eine 16-Bit-Adresse angesprochen werden. Bei diesen Ein-Byte-Befehlen enthält der Operationscode das Registerpaar, das auf eine Adresse zeigt, dessen Inhalt transportiert oder verarbeitet werden soll.

Der Vorteil dieser Adressierart besteht darin, daß ein Datentransport aus dem Speicher oder in den Speicher durch einen kurzen Ein-Byte-Befehl veranlaßt werden kann. Da der Inhalt des auf den Prozessorchip untergebrachten Registerpaars sich leicht ändern läßt, wird durch diese

Adressierart etwa das gleiche erreicht, wie es bei anderen Typen durch Index- bzw. indirekte Adressierung möglich ist.

3.8.3 Unmittelbare Adressierung

Wenn ein beliebiges 8-Bit-Muster, das in jeder gewünschten Weise, also als konstanter Zahlenwert, als Symbol für einen Buchstaben oder zur Darstellung gewisser Schaltzustände definiert werden kann, dem Prozessor zur weiteren Verarbeitung zugeführt werden soll, verwendet man die *unmittelbare* (IMMEDIATE) *Adressierung*. Bei dieser Adressierart folgt dem ersten Befehlsbyte, das den Befehlscode und das Zielregister enthält, immer ein zweites Befehlsbyte mit der Information, die in das Zielregister geladen werden soll (Bild 3.14).

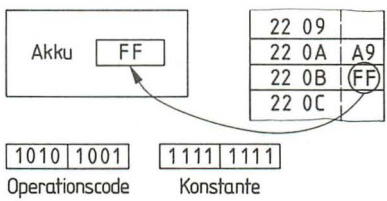


Bild 3.14 Unmittelbare Adressierung
Es handelt sich im allgemeinen um Zwei-Byte-Befehle. Im ersten Byte steht der Operationscode, der auch das Quellen- oder Zielregister enthält. Das zweite Befehlsbyte enthält einen Binärwert, der als Konstante unmittelbar durch den Befehl verarbeitet wird.

Bei vielen Prozessoren kann auch eine Datenverarbeitung durch unmittelbare Adressierung erfolgen. In diesem Fall steht der erste Operand in dem im Operationscode angezogenen Register, und der Inhalt des zweiten Befehlsbyte stellt den zweiten Operanden dar. Das Ergebnis der Datenverarbeitung erscheint dann auch in dem Register, in dem vorher der erste Operand gestanden hat.

3.8.4 Absolute Adressierung

Leicht verständlich ist auch die Wirkungsweise der *absoluten Adressierung*. Befehle mit dieser Adressierungsart bestehen immer aus drei Byte. Das erste Byte enthält wiederum den Operationscode, der das Zielregister

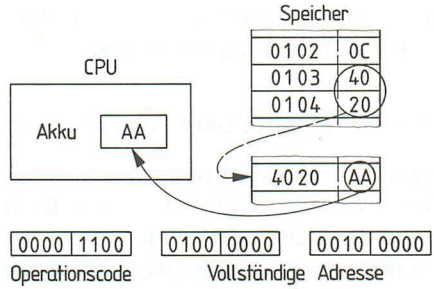


Bild 3.15 Absolute Adressierung
Bei absoluter Adressierung werden Drei-Byte-Befehle verwendet, bei denen im zweiten und dritten Befehlsbyte die vollständige Operationsadresse steht.

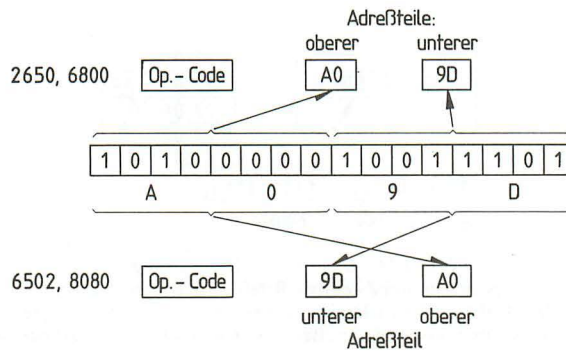


Bild 3.16 Verschiedene Reihenfolge der Adreßteile bei absoluter Adressierung
Die Anordnung der beiden Teile der Adresse im zweiten und dritten Befehlsbyte ist bei den verschiedenen Mikroprozessor-Typen unterschiedlich. Während z.B. beim *2650* und beim *6800* der obere Adreßteil im zweiten und der untere Adreßteil im dritten Byte stehen, ist die Reihenfolge beim *6502* und beim *8080* genau umgekehrt.

oder das Register des ersten Operanden mitenthalten muß. Im zweiten und dritten Byte steht dann die vollständige Operandenadresse, für die eine Kapazität von 16 Bit notwendig ist (Bild 3.15).

Bei der absoluten Adressierung gibt es leider einen grundlegenden Unterschied zwischen den Mikroprozessor-Typen, der zu erheblicher Verwirrung führen kann, wenn man die für verschiedene Typen geschriebenen Programme betrachtet.

Da für eine Adresse im allgemeinen 16 Bit benötigt werden, die auf zwei 8 Byte breite Befehlsworte zu verteilen sind, muß man die Adresse auch in einen *oberen* und in einen *unteren Adreßteil* aufteilen. Bei einer Reihe von Prozessoren enthält nun das zweite Befehlsbyte den oberen Adreßteil, den man auch *höherwertigen Adreßteil* nennt, und das dritte Befehlsbyte den unteren Adreßteil, der auch *niederwertiger Adreßteil* heißt. Auch im Befehl steht also die Adresse in der richtigen Reihenfolge mit dem oberen Adreßteil links vom unteren (Bild 3.16).

Bei anderen Prozessoren muß im Befehl der untere Adreßteil vor dem oberen Adreßteil geschrieben werden. Das zweite Befehlsbyte enthält darum also den unteren und das dritte Befehlsbyte den oberen Adreßteil. Beim Schreiben eines Programms, das in dieser Weise formatiert werden muß, kann es leicht Irrtümer geben, aber man kann sich natürlich auch hieran gewöhnen.

3.8.5 Seite-Null-Adressierung

Da die absolute Adressierung immer drei Befehlsbyte und somit auch drei Speicherplätze benötigt, kennen einige Prozessoren noch eine Adressierungsart, bei der im zweiten Befehlsbyte nur eine 8-Bit-Adresse enthalten ist. Mit diesen Befehlen kann man natürlich nur 256 verschiedene Speicherplätze ansprechen, und man muß dem Prozessor noch zusätzlich sagen, in welchem Teil des Gesamtspeicherbereiches diese 256 Speicherplätze zu finden sind. Im allgemeinen wählt man dafür die *Seite*

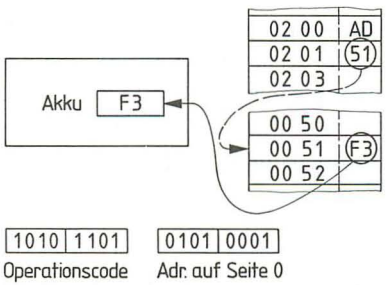


Bild 3.17 Seite-Null-Adressierung
Da bei diesen Befehlen der anzusprechende Adreßraum nur in der Seite Null, also von hexadezimal **0000** bis **00FF**, liegt, genügt das zweite Befehlsbyte für die Angabe der Adresse.

Null (PAGE ZERO), d.h. den *ersten Anfang des Speicherbereichs* mit den Adressen **0000** bis **00FF** (siehe auch Bild 1.12). Bei dieser Adressierung enthalten die acht Bit des oberen Adreßteils jeweils eine **0**, was aber in dem Befehl selbst nicht zum Ausdruck gebracht werden muß, da der Prozessor Befehle mit Seite-Null-Adressierung aus dem Operationscode selbst erkennt (Bild 3.17).

Leider verwenden nun einige Mikroprozessor-Hersteller abweichende Bezeichnungen für die absolute und die Seite-Null-Adressierung. Sie sprechen von EXTENDED ADDRESSING, wenn absolute Adressierung gemeint ist, und von DIRECT ADDRESSING, wenn nur der Adressenbereich auf Seite Null angesprochen wird.

3.8.6 Relative Adressierung

Bei dieser Adressierart muß der Prozessor die gültige Adresse, man spricht auch von der *effektiven Adresse* (EFFECTIVE ADDRESS), durch einen Rechenvorgang finden. Wie das funktioniert, ist in Bild 3.18 dargestellt. Grundsätzlich werden bei *relativer Adressierung* Befehle mit zwei Byte Länge benötigt. Das erste Byte enthält wiederum den Operationscode, während im zweiten Byte eine Binärzahl im Zweier-Komplement, also eine Dezimalzahl zwischen +127 und −128, enthalten sein muß. Diese Zahl kennzeichnet die *relative Verschiebung,* englisch OFFSET, im Verhältnis zur nächstfolgenden Befehlsadresse.

> Bei relativer Adressierung wird zu der nach Befehlsausführung im Befehlsadreß-Register stehenden Adresse die im Befehl stehende relative Verschiebung nach den Regeln der Zweier-Komplementarithmetik addiert. Das Ergebnis ist wieder eine Adresse, in der der Prozessor den zu verarbeitenden Operanden findet, oder die er als Sprungziel verwendet.

Das soeben Gesagte sei im Beispiel Bild 3.18 noch einmal erklärt.

In den Adressen **1202** und **1203** befindet sich der Befehl mit dem Code **C8 F9.** Der Inhalt des ersten Befehlsbyte, also **C8,** ist der Befehlscode und der des zweiten, **F9,** die relative Verschiebung. **F9** entspricht der negati-

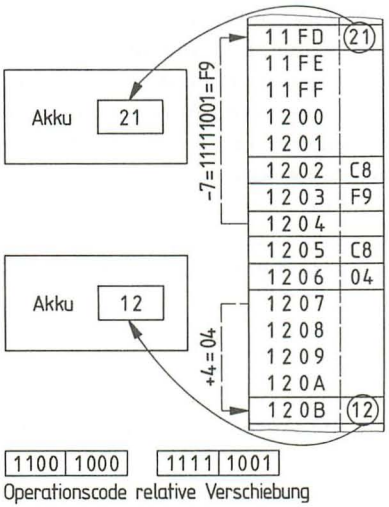


Bild 3.18 Relative Adressierung
Bei der relativen Adressierung wird die effektive Adresse, d.h. die Adresse, in der der Operand steht, durch einen Rechenvorgang gefunden. Grundsätzlich wird das zweite Befehlsbyte vom Prozessor als Zahl im Zweier-Komplementsystem interpretiert und zum Stand des Befehlsadreß-Registers addiert. Das Ergebnis ist die als nächstes zu verarbeitende Adresse, die bei voller Ausnutzung des zweiten Befehlsbyte für diese relative Verschiebung + 127 und – 128 Adressen von der dem Befehl mit relativer Adressierung als nächstes folgenden Befehlsadresse entfernt liegen kann.

ven Dezimalzahl **7**, was bedeutet, daß der Prozessor den Operanden **7** Speicherstellen zurückliegend zu suchen hat. Die Zählung beginnt mit der dem Befehl folgenden Speicherstelle, also bei **1204.** Wie in der Elektronik üblich, fängt man aber auch hier nicht mit **1**, sondern mit **0** an zu zählen, so daß der Operand in der Speicherstelle **11FD** zu finden ist.

Das gleiche gilt für den zweiten, auf dem Bild dargestellten Befehl. Dieser ist in den Speicherstellen **1205** und **1206** gespeichert und enthält im zweiten Befehlsbyte die relative Verschiebung **04**, was der positiven Dezimalzahl **4** entspricht. In diesem Falle muß also von der dem Befehl folgenden Speicherstelle aus, also von **1207** ausgehend, wieder mit **0** beginnend vorwärts gezählt werden, so daß in diesem Fall der Operand in der Speicherstelle **120B** steht.

Nicht alle Mikroprozessoren kennen die relative Adressierung in

ihrem Befehlssatz. Bei einigen Prozessoren ist relative Adressierung nur bei Sprungbefehlen möglich, und andere wiederum erlauben diese Befehlsart auch für Datentransport- und Datenverarbeitungs-Befehle.

Der große Vorteil der relativen Adressierung besteht darin, daß Programmabschnitte, bei denen nur relative und ggf. noch Register- und unmittelbare Adressierung vorkommen, beliebig im Speicherraum verschoben werden können, ohne daß eine Änderung der im Programm angezogenen Adressen notwendig ist. Man nennt diese Programmabschnitte darum auch verschiebbar, englisch RELOCTABLE.

3.8.7 Indirekte Adressierung

Das Kennzeichnende dieser Adressierart ist es, daß im Operandenteil des Befehls nicht sofort die Adresse, in der der Operand zu finden ist, steht.

> Bei *indirekter Adressierung* ist im Befehl nur die Adresse zu finden, in der die Adresse steht, die erst den wirklichen Operanden enthält.

Ein Beispiel für diese Adressierart ist in Bild 3.19 dargestellt. Der Drei-Byte-Befehl in den Speicherstellen **2100, 2101** und **2102** enthält wiederum im ersten Byte den Operationscode, der auch die Anweisung für indirekte Adressierung enthalten muß. Das zweite und das dritte Befehlsbyte enthält eine Adresse in absoluter Darstellung, im Beispiel die Adresse **4000.** Da diese Adresse in zwei Byte dargestellt ist, kann sie in einem beliebigen Teil des Speicherraums liegen.

In der im Befehl enthaltenen indirekten Adresse **4000** kann aber, da in einer Adresse nur 8 Bit abgespeichert werden können, auch nur ein Teil der Operandenadresse enthalten sein. So wird grundsätzlich bei indirekter Adressierung noch die der indirekten Adresse folgende Speicherstelle für die Abspeicherung der vollständigen Operandenadresse mitverwendet. In den Speicherstellen **4000** und **4001** findet man im Beispiel die Adresse **8049.** In dieser Adresse befindet sich dann erst der eigentliche Operand, der in dem als Beispiel dargestellten Befehl in den Akkumulator transportiert wird.

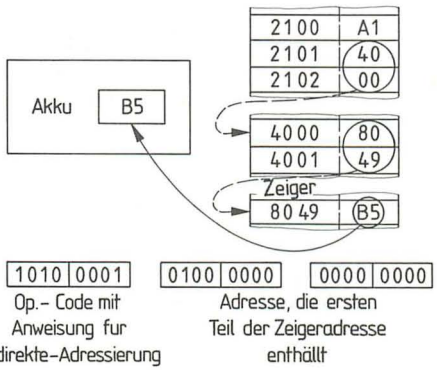


Bild 3.19 Indirekte Adressierung
Bei indirekter Adressierung enthält der Befehl noch nicht die eigentliche Operandenadresse, sondern dort steht zunächst nur wiederum eine Adresse, die erst auf die eigentliche effektive Adresse, in der der zu verarbeitende Operand steht, zeigt. Da für eine Adresse 16 Bit benötigt werden, enthält die im Befehl stehende Adresse nur einen Teil des Zeigers und die nachfolgende Adresse dann den zweiten Teil, wobei sich die Reihenfolge der Adressenteile bei den einzelnen Prozessortypen, wie im Bild 3.16 gezeigt, unterscheidet.

Der große Vorteil der indirekten Adressierung besteht nun darin, daß die *indirekte Adresse* auch als *Zeiger,* der auf eine andere Adresse zeigt, verwendet werden kann. So ist z.B. möglich, den Zeiger immer auf eine ganz bestimmte RAM-Speicherstelle zeigen zu lassen. Durch Programm kann man dann in diese Speicherstelle beliebige Daten laden, die dann durch den Befehl mit der indirekten Adressierung verarbeitet werden.

3.8.8 Index-Adressierung

Sehr effektiv für die Verarbeitung von *Datenblöcken* oder *Datenfeldern,* worunter man eine Anzahl von hintereinander im Speicher enthaltener Informationen versteht, ist die Index-Adressierung. Sie ist allerdings nicht in den Befehlssätzen aller Mikroprozessor-Typen enthalten.

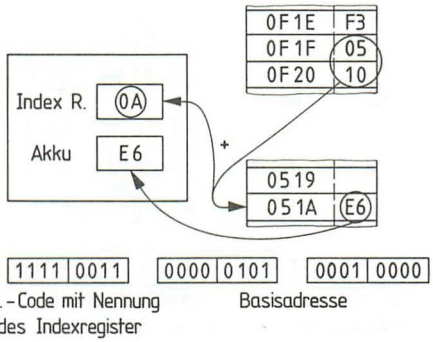


Bild 3.20 Index-Adressierung
Auch bei der Index-Adressierung wird die effektive Adresse, in der der Operand zu finden ist, erst durch einen Rechenvorgang bestimmt. Hierbei wird die im Befehl enthaltene Basisadresse und der Inhalt des Indexregisters addiert, wodurch sich die Operandenadresse ergibt.

Das Prinzip der *Index-Adressierung* besteht darin, daß der Prozessor die effektive Adresse, d.h. die Adresse, in der der Operand zu finden ist, vor der Befehlsverarbeitung durch eine Additionsrechnung ermittelt. Bei dieser Addition wird jeweils der Inhalt des Indexregisters zum Inhalt einer im Befehl enthaltenen Basisadresse hinzugezählt.

Ein Beispiel ist in Bild 3.20 dargestellt. Das erste Byte des Befehls enthält wiederum den Operatinscode, aus dem zu ersehen sein muß, daß mit Index-Adressierung gearbeitet werden soll. Das zweite und dritte Befehlsbyte ist in diesem Fall als Basisadresse in absoluter Darstellung zu interpretieren. Im Beispiel ist die Basisadresse **0510** und der Inhalt des Indexregisters **0A.** Die Addition dieser beiden Werte ergibt die effektive Adresse **051A,** in der der Operand für den Ladebefehl an den Akkumulator zu finden ist.

Bei einigen Prozessoren ist das Indexregister mit einer größeren Kapazität aufgebaut und kann mit einer vollständigen 16-Bit-Adresse geladen werden. In diesem Fall befindet sich also die Basisadresse im Indexregister, und die Abweichung von der Basisadresse steht im Befehl. Da die effektive Adresse im Adreßraum von der Basisadresse nur maximal 255

Speicherplätze entfernt liegen darf, braucht man bei diesen Prozessoren nur zwei Befehlsbyte, denn im zweiten Byte lassen sich die Dezimalzahlen von 0 bis +255 binär darstellen.

Für viele Verwendungszwecke sehr nützlich ist es, daß einige Prozessoren es erlauben, den Inhalt des Indexregisters vor Ausführung eines Befehls mit Index-Adressierung um den Wert 1 zu erhöhen oder zu erniedrigen. Man nennt dies *Autoincrement* bzw. *Autodecrement*. Man erreicht hierdurch, daß innerhalb einer Programmschleife bei jedem Schleifendurchgang der Operand aus einer anderen Speicherstelle geholt wird. Bei Prozessoren, die die selbsttätige Incrementierung und Decrementierung nicht kennen, muß die Erhöhung oder Erniedrigung des Indexregisterinhalts durch zusätzliche Befehle erfolgen.

Es gibt dann noch bei einigen Prozessoren Befehle, bei denen die indirekte und die Index-Adressierung kombiniert werden. Hierauf wird bei der Besprechung der Befehlssätze in Kapitel 4 noch eingegangen.

3.9 Ein- und Ausgabeoperationen

Als *Eingabe* bzw. *Ausgabe* bezeichnet man die Befehlsabläufe, bei denen ein Mikrocomputer Informationen nach außen abgibt bzw. von dort einholt. Mit diesen Ein-/Ausgabeoperationen (INPUT/OUTPUT OPERATIONS) können durch entsprechende Bedienungselemente oder auch durch selbsttätig arbeitende *Sensoren,* d.h. Meßwertaufnehmer, Informationen in das Mikroprozessor-System eingegeben werden, beziehungsweise kann das System hiermit Daten an sogenannte *periphere Geräte* ausgeben, die wir Menschen dann ablesen oder sonst irgendwie wahrnehmen können.

Nun kennen nicht alle Mikroprozessoren in ihrem Befehlssatz *besondere Ein- und Ausgabebefehle*. Denn es ist ohne weiteres möglich, statt einer bestimmten Speicherstelle einen an diese Adresse angeschlossenen *Ein-/Ausgabebaustein* zu adressieren. Dieser Baustein muß dann natürlich auch Anschluß an den Datenbus haben. Wenn seine Adresse durch den Prozessor aufgerufen wird, übernimmt er das auf dem Datenbus anstehende Bitmuster in einen Zwischenspeicher und stellt die so empfangene Information, ggf. nach Verstärkung, für eine weitere Verarbeitung außerhalb des eigentlichen Mikrocomputers zur Verfügung.

In ähnlicher Weise kann ein derartiger Baustein auch für den Daten-

eingang verwendet werden. Hier werden die von außen kommenden Signale so lange zwischengespeichert, bis der Baustein über seine Adressenleitung vom Prozessor angesprochen wird. Die zwischengespeicherte Information gelangt dann auf den Datenbus und damit zur Verarbeitung in das Computersystem.

Man bezeichnet nun eine derartige *Ein- und Ausgabestelle* mit dem englischen Wort PORT und versteht darunter eine *Schnittstelle zum Datenein- oder -ausgang* mit mehreren parallelen Leitungen. Man unterscheidet zwischen *Eingangs-* und *Ausgangsport*. Es gibt aber auch Prozessoren, die *bidirektionale Port,* die in beiden Richtungen arbeiten können, bedienen. Um den Datenverkehr der Peripherie zu erleichtern, haben einige Prozessor-Typen besondere Ein- und Ausgabebefehle mit nur ein oder zwei Byte Länge in ihrem Befehlssatz. Es werden zum Teil die gleichen Adreßbezeichnungen wie bei den Speicheradressen verwendet, so daß diese Prozessoren dann auf einer besonderen Steuerleitung noch ein Signal ausgeben, aus dem zu erkennen ist, ob eine Speicher- oder eine Ein-/Ausgabeadresse angesprochen wird.

Alle Hersteller von Mikroprozessoren haben in ihrem Programm auch besondere hochintegrierte Bausteine, in denen alle Hardwarekomponenten, die für Aus- und Eingabevorgänge notwendig sind, zusammengefaßt werden. Je nach Hersteller gibt es hierfür eine Menge verschiedener Bezeichnungen. Im wesentlichen unterscheidet man jedoch Bausteine, die eine *parallele Datenausgabe* erleichtern, und solche, die für eine *serielle Datenausgabe* vorgesehen sind (Bild 3.21).

Diese Bausteine lassen es zu, daß durch Programmbefehle eine ganz bestimmte Arbeitsweise festgelegt wird. Ebenso können auch durch Programmierung bestimmte Leitungen als Ein- oder Ausgabeleitungen geschaltet werden. Bei den Bausteinen für serielle Datenein- und -ausgabe ist es nötig, die Geschwindigkeit, mit der der Datentransport erfolgen soll, durch Vorgabe der *Baudrate* zu bestimmen. Hierbei entspricht ein Baud einer Übertragungsgeschwindigkeit von einem Bit pro Sekunde. Die Baudrate ist also ein Maß für die Schnelligkeit des Datentransports bei serieller Übertragung.

Im allgemeinen wird bei Ein- und Ausgabeoperationen im *Quittungsverkehr,* englisch HANDSHAKE MODE, gearbeitet. Dies bedeutet, daß der Prozessor immer dann, wenn er Daten zur Ausgabe bereitstellt, auf einer besonderen Steuerleitung ein Signal aussendet, das dem Empfangsgerät das Vorhandensein von gültigen Daten anzeigt. Auf einer weiteren

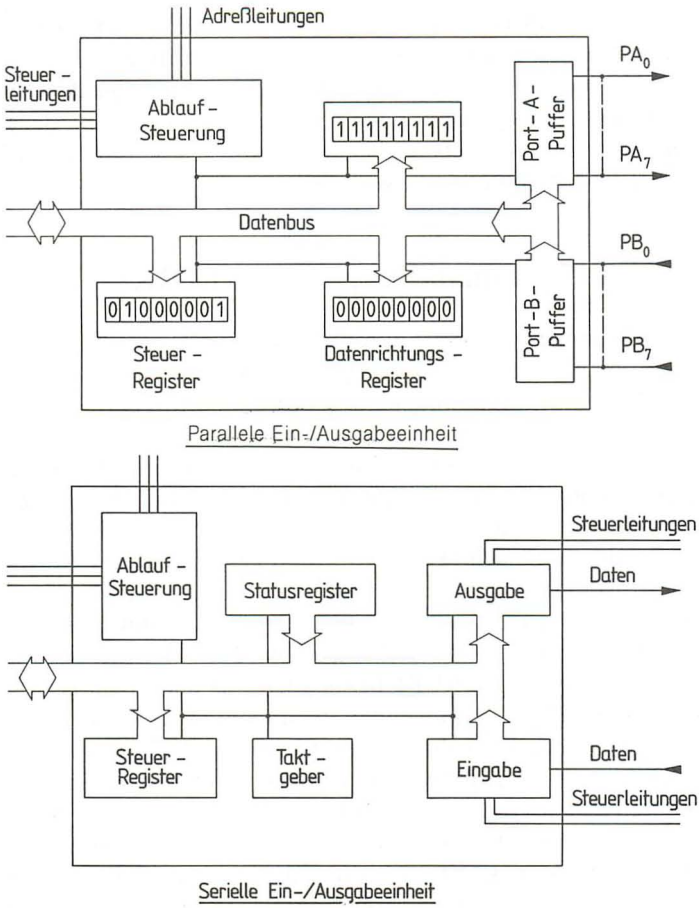


Bild 3.21 Ein- und Ausgabeeinheiten

Für alle Mikroprozessortypen gibt es spezielle Ein- und Ausgabeeinheiten, die mit den verschiedensten Namen bezeichnet werden. Man unterscheidet:

Parallele Ein-/Ausgabeeinheiten, z.B. PIO (PARALLEL INPUT OUTPUT DEVICE), PIA (PARALLEL INTERFACE ADAPTER) und serielle Ein-/Ausgabeeinheiten, z.B. USART (UNIVERSAL SYNCHRONOUS RECEIVER TRANSMITTER), ACIA (ASYNCHRONOUS COMMUNICATION INTERFACE ADAPTER).

Diese Bausteine müssen meist zunächst durch Steuerbefehle aktiviert werden, wobei die Datenkanäle dann wahlweise als Eingang oder Ausgang geschaltet werden können. Die seriellen Ein-/Ausgabeeinheiten besitzen meist einen eigenen Taktgenerator, durch den die serielle Ausgabe der vom Prozessor in paralleler Form übernommenen Daten mit einer genau festgelegten Geschwindigkeit geregelt wird.

Steuerleitung muß nun das angesprochene Gerät wiederum mitteilen, daß es zur Datenübernahme bereit ist und durch Änderung des Pegels auf dieser Leitung die Beendigung der Datenübernahme signalisieren.

Da diese Steuersignale für die Programmierung keine Bedeutung haben, sondern durch entsprechende Schaltung, also durch Hardware, bestimmt sind, soll hier nicht weiter darauf eingegangen werden.

3.10 Behandlung von INTERRUPT

Normalerweise arbeitet ein Mikroprozessor sein Programm in der Reihenfolge ab, wie es ihm bei der Programmierung vorgegeben wurde. Innerhalb dieses Programmes kann er z.B. Daten von Peripheriegeräten abfragen und diese dann anschließend verarbeiten. Bei einem so organisierten Programm müssen die angeschlossenen Geräte ihre Daten also so lange bereithalten, bis die Abfrage ihres Gerätes in der Programmfolge an der Reihe ist.

Es gibt aber nun auch in allen Mikroprozessor-Systemen Möglichkeiten, in den Programmablauf von außen einzugreifen und dadurch den Mikroprozessor zu einem Sprung in ein ganz bestimmtes Unterprogramm zu veranlassen. Dieser Vorgang wird im allgemeinen mit dem englischen Wort INTERRUPT bezeichnet, was man nicht ganz zutreffend auch mit *Unterbrechung* übersetzen kann (Bild 3.22).

Die Mikroprozessoren haben meist in ihrem Befehlssatz auch eine Möglichkeit, die es dem Programmierer erlaubt, bei bestimmten Programmteilen entweder eine *Unterbrechung zuzulassen* (ENABLE INTERRUPT) oder den *Interrupt zu sperren* (DISABLE oder INHIBIT INTERRUPT). Man nennt das Sperren des Interrupts gelegentlich auch *Maskieren,* und einige Prozessoren haben eine besondere Interruptanforderungsleitung, die sich nicht sperren läßt (NONMASCABLE INTERRUPT).

Grundsätzlich führt der Prozessor, ohne daß hierfür weitere Programmbefehle notwendig sind, bei einer Interruptanforderung von außen, wenn der Interrupt nicht gesperrt ist, folgende Operationen durch:

- Der Prozessor meldet dem anfordernden Gerät, daß die Unterbrechungsanforderung angenommen wurde.
- Der zur Zeit laufende Befehl wird beendet.

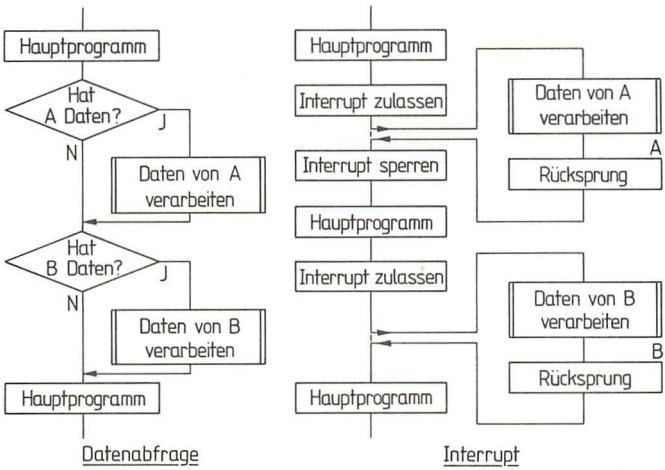


Bild 3.22 Methoden der Datenübernahmen von einem Peripheriegerät
Datenübernahmen in ein Mikroprozessor-System können im Prinzip entweder durch zyklische Abfragen oder durch erzwungene Programmunterbrechung (Interrupt) erfolgen.
Bei der Datenabfrage-Methode müssen im Programm entsprechende Befehle stehen, durch die immer wieder gefragt wird, ob an dem Peripheriegerät Daten zur Übernahme bereitstehen. Ist dies der Fall, werden sie übernommen und verarbeitet.
Bei der Interrupt-Methode kann jedes Peripheriegerät von sich aus die Übernahme seiner Daten verlangen. Soweit der Interrupt nicht durch Programmbefehle gesperrt ist, unterbricht der Prozessor, wenn eine Interrupt-Anforderung vorliegt, sein derzeit laufendes Programm, übernimmt z.B. die Daten und kehrt dann wieder zum Hauptprogramm zurück.

- Wie beim Anruf eines Unterprogramms wird die dem laufenden Befehl folgende Befehlsadresse im Stapelspeicher abgespeichert.
- Der Prozessor stellt fest, welches Unterprogramm durch den Interrupt angerufen wurde, und lädt die erste Befehlsadresse dieses Programms in das Befehlsadreß-Register.
- Das Unterprogramm wird ausgeführt, bis dort der Rücksprungbefehl erscheint.
- Die vorher im Stapelspeicher abgelegte Adresse des im Hauptprogramm folgenden Befehls wird wieder in das Befehlsadreß-Register geladen und das Hauptprogramm mit diesem Befehl fortgesetzt.

Im Grundsatz ist es möglich, auch mehrere durch Interrupt ausgelöste Unterprogramme ineinander zu verschachteln oder aber den Interruptanforderungen verschiedener peripherer Geräte eine unterschiedliche Priorität bezüglich Behandlung ihrer Unterbrechungsanforderung zu geben.

4

Die Befehlssätze der Mikroprozessoren 2650, 6502, 6800 und 8080/85*

Wer einen Mikroprozessor programmieren will, muß seinen Befehlssatz beherrschen und die Wirkung der Befehle kennen. Leider gibt es noch keine einheitliche Assemblersprache. Jeder Hersteller von Mikroprozessoren folgt hierbei seinen eigenen Vorstellungen.

In diesem Kapitel werden die Befehlssätze von vier weitverbreiteten 8-Bit-Mikroprozessoren, die universell verwendbar sind, behandelt. Die Wirkung der Befehle wird im einzelnen erklärt und miteinander verglichen. Da die Befehlssätze dieser Prozessoren im wesentlichen alle Möglichkeiten enthalten, die die Mikroprozessor-Technik überhaupt bieten kann, dürfte es nach dem Studium dieses Abschnitts nicht schwerfallen, auch die Befehlssätze anderer Typen zu verstehen.

Für die Programmierarbeit mit einem guten Entwicklungsgerät reicht es, wenn der Programmierer die Assemblersprache beherrscht. Der Anfänger kann aber die Wirkung der Befehle viel besser verstehen, wenn diese in der Maschinensprache dargestellt werden. In den folgenden Abschnitten werden darum alle Befehle sowohl in Assemblersprache als auch in Maschinensprache hexadezimal codiert angegeben.

---

* Bei allen Mikroprozessoren gibt es sogenannte Typenfamilien; das heißt, verschiedene Mikroprozessoren sind im Aufbau ähnlich und haben im wesentlichen den gleichen Befehlssatz. Man spricht hierbei von einer Kompatibilität der Befehle und versteht hierunter, daß ein Programm, das für einen Prozessor einer Familie geschrieben ist, ohne größere Änderungen auch auf einem anderen Prozessor der gleichen Familie laufen kann. Dies bedeutet also, daß die in diesem Kapitel jeweils für einen Prozessor gegebenen Erklärungen auch für die anderen Prozessoren der gleichen Familie gelten.

4.1 Befehlstabellen und dort verwendete Symbole

Die Hersteller von Mikroprozessoren liefern ihren Kunden Befehlstabellen, die fast ausschließlich in englischer Sprache abgefaßt sind. Dies ist nicht so schlimm, denn es werden vielfach Symbole verwendet, und eine gewisse Kenntnis der englischen Sprache ist für alle, die etwas tiefer in die Mikroprozessor-Technik eindringen wollen, notwendig.

Bedauerlich ist es nun allerdings, daß diese Befehlstabellen in der Praxis nur schwer zu handhaben sind, denn Übersichtlichkeit und Druckausführung kann bei manchen Tabellen nur als schlecht bezeichnet werden.

Nachteilig ist ferner, daß die einzelnen Hersteller verschiedene Ausdrücke der englischen Sprache für den gleichen Begriff verwenden und daß man sich vor allem bezüglich der Symbole, mit denen Befehle in kurzer Form beschrieben werden, nicht einig ist.

Es ist darum durchaus zu empfehlen, wenn sich Programmieranfänger eigene Befehlstabellen herstellen, die dann alles das enthalten sollen, was beim Programmieren gebraucht wird. Hierbei kann man sehr viel lernen, muß man doch Aufbau und Wirkung jedes einzelnen Befehls anhand des Programmierhandbuches, das die Hersteller meist auch nur in englischer Sprache liefern, genau ergründen, um den Befehl für die eigene Tabelle richtig beschreiben zu können.

⟶	Datentransportrichtung
⟷	Datentausch
( )	Inhalt einer Speicherstelle od. eines Registers
(( ))	Verschiedene Darstellungen
#, @, •	für indirekte Adresse (Zeiger)
$\overline{\text{AKKU}}$	Komplementierung
Λ oder •	Und = AND
V oder +	Oder = OR
∀ oder ⊕	Nur oder = EXCLUSIV OR
R	Register
M od. Adr	Inhalt eines Speicherplatzes
PC od. $	Befehls - Adreß - Register (PROGRAM COUNTER)
SP	Stapelspeicher - Zeiger (STACK POINTER)

Bild 4.1 Wichtige Symbole für Befehlstabellen

Die Symbole werden zur Erläuterung der Befehlswirkung in den Befehlstabellen der Mikroprozessor-Hersteller verwendet. Leider sind sie nicht ganz einheitlich, da für den gleichen Vorgang oft verschiedene Symbole Verwendung finden.

Für eine kurze Beschreibung von Befehlen eignen sich gewisse Symbole, die im Bild 4.1 dargestellt sind. Diese Symbole werden auch bei der Einzelbeschreibung der Befehle im folgenden Abschnitt verwendet, wobei einige der Symbolzeichen auch für die verschiedenen Assemblersprachen benötigt werden. Leider gibt es auch hier keine vollständige Übereinstimmung. Als Zeichen für indirekte Adressierung werden z.B. die verschiedensten Symbole verwendet. Bei den folgenden Befehlsbeschreibungen und Programmbeispielen werden immer die Symbole eingesetzt, die für die Assemblersprache des betreffenden Prozessors vorgeschrieben sind.

4.2 Durch Befehle zugängliche Register

Die Befehlssätze der einzelnen Mikroprozessoren werden durch deren inneren Aufbau bestimmt. Für den Programmierer ist es dabei wichtig, zu wissen, welche CPU-Register er durch Befehle beeinflussen kann. Wie Bild 4.2 zeigt, ist die Struktur der in diesem Abschnitt behandelten Prozessoren recht unterschiedlich, deutlich kann man zwei verschiedene Arten des Aufbaus unterscheiden.

Der *6502* und der *6800* besitzen jeweils nur wenige Register, auf die der Programmierer direkt Zugriff hat. Demgegenüber haben die Typen *2650* und *8080* eine ganze Reihe von Hilfsregistern in der CPU, die durch Befehle ansprechbar sind.

Akkumulatoren gibt es in allen Prozessoren. *8080* und *6502* haben einen Akku, während der *6800* zwei Stück besitzt. Etwas anders ist der Aufbau beim *2650,* der neben dem Register 0, das einem Akkumulator entspricht, noch zweimal drei Vielzweckregister hat, die im wesentlichen in gleicher Weise benutzt werden können wie ein Akkumulator. Je nach einer Anweisung im Programm-Status-Wort können entweder die Register 1, 2 und 3 oder die Register 1′, 2′ und 3′ im Programm arbeiten.

Alle Prozessoren haben auch ein Befehls-Adreß-Register (PROGRAM COUNTER = PC) und Programm-Status-Register = PSR. Üblicherweise hat das PC-Register eine Kapazität von 16 Bit und das Programm-Status-Register, das beim *6800* CONDITION CODE REGISTER und bei *8080* FLAG REGISTER genannt wird, 8 Bit. Abweichend von den anderen Prozessoren kann der *2650* nur 32 768 Speicherstellen adressieren und braucht darum für das Befehls-Adreß-Register nur 15 Bit.

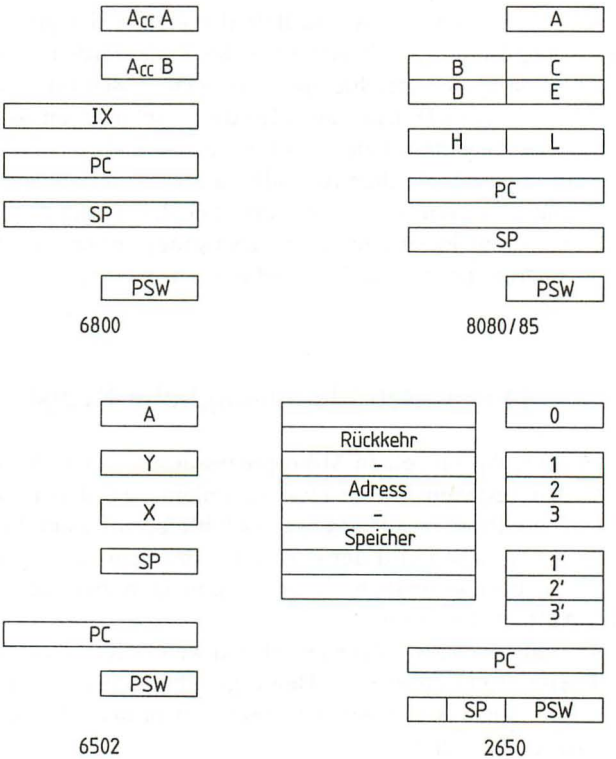


Bild 4.2 Durch Programmbefehle zugängliche Register der Mikroprozessoren 6800, 8080/85, 6502 und 2650

Die verschiedenen Mikroprozessortypen unterscheiden sich grundsätzlich im Aufbau der Zentraleinheit bezüglich der dort ansprechbaren Register.

Der *6800* besitzt zwei 8-Bit breite Akkumulatoren mit der Bezeichnung A und B sowie ein 16-Bit-Indexregister (IX), ein 16-Bit-Befehlsadreß-Register (PROGRAMCOUNTER) und ein 16-Bit-Stapelzeiger-Register (STACK POINTER). Das Programm-Status-Register hat 8 Bit.

Die Prozessoren *8080/85* besitzen einen Akkumulator (Register A) und sechs Hilfsregister mit den Bezeichnungen B, C, D, E, H und L, die auch zu drei Registerpaaren zusammengefaßt werden können. Befehlsadreß-Register (PC) und Stapelzeiger (ST) haben 16 Bit, das Programm-Status-Wort 8 Bit.

Der *6502* besitzt einen 8-Bit-Akkumulator und zwei 8-Bit-Indexregister, die X und Y genannt werden. Stapelzeiger und Programm-Status-Wort-Register haben 8 Bit und das Befehlsadreß-Register 16 Bit Inhalt.

Der *2650* besitzt insgesamt sieben Vielzweckregister, von denen das Register 0 immer verfügbar ist. Sechs andere Register sind in zwei Registerbänken angeordnet. Durch Programmbefehl kann entweder die Registerbank 0 mit den Registern 1, 2 und 3 oder die Registerbank 1

mit den Registern 1', 2' oder 3' angerufen werden. Das Befehlsadreß-Register hat eine Kapazität von 15 Bit; es gibt zwei Programm-Status-Register. Im Baustein selbst integriert ist der Rückkehradreß-Speicher, der sechs 15-Bit-Adressen für den Rücksprung aus Unterprogrammen aufnehmen kann. Der dazu gehörige Stapelzeiger befindet sich im oberen Programm-Status-Wort.

Dafür besitzt er zwei Programm-Status-Register, die mit PSL = PROGRAM STATUS LOWER (unterer Programm-Status) und PSU = PROGRAM STATUS UPPER (oberer Programm-Status) bezeichnet werden.

Abweichend von den anderen Prozessoren besteht beim *8080* die Möglichkeit, jeweils zwei der 6 Hilfsregister zu einem Registerpaar zusammenzufassen. Dabei spielt das Registerpaar HL noch eine besondere Rolle, da sein Inhalt bei bestimmten Befehlen auch als 16-Bit-Adresse verwendet werden kann.

Dafür hat der *2650* wieder eine andere Besonderheit. Er hat den Stapelspeicher, den STACK, auf dem Prozessorchip, und nicht wie bei den anderen Prozessoren, im Speicherbereich. Dieser Stapelspeicher wird benutzt, um beim Anruf von Unterprogrammen die Rückkehr-Adresse zu speichern. Der *2650* braucht darum auch nur einen 3-Bit-Stapelzeiger, der im oberen PSW untergebracht ist, denn es werden maximal 8 Rückkehr-Adressen gespeichert.

6800 und *8080* haben einen 16-Bit-Stapelzeiger, so daß der Stapelspeicher beliebig im Adreßraum angelegt werden kann. Der 8-Bit-Stapelzeiger des *6502* erlaubt die Unterbringung des Stapelspeichers nur auf der Seite Eins des Adreßraums.

6502 und *6800* haben zwei bzw. ein Indexregister. Beim *2650* können alle Vielzweckregister auch als Indexregister verwendet werden, während beim *8080* das Registerpaar HL eine ähnliche Funktion wie ein Indexregister erfüllen kann.

Die Prozessoren *2650, 6502* und *6800* können auch mit indirekter Adressierung im üblichen Sinne arbeiten, was nicht aus dem Bild 4.2 zu erkennen ist. Beim *8080* muß für eine Art indirekter Adressierung das Registerpaar HL verwendet werden.

Eine Kombination von Index- und indirekter Adressierung ist beim *2650,* beim *6502* und beim *6800* möglich, jedoch gibt es dabei noch gewisse Unterschiede, die später noch erklärt werden.

4.3 Datentransport-Befehle

4.3.1 Ein-Byte-Befehle mit Register-Adressierung

Die meisten Mikroprozessoren haben in ihrem Befehlssatz eine Reihe von kurzen, nur ein Byte langen Befehlen für den Datentransport zwischen den CPU-Registern. Der Befehl enthält in diesem Fall im Operationscode in codierter Form auch die beiden Register, die am Datentransport als Datenquelle bzw. als Datenziel beteiligt sind.

Prozessoren, die in ihrer CPU nur wenige gleichwertige, das heißt in diesem Fall 8-Bit-Register haben, kennen nur wenige dieser kurzen Transferbefehle. Dementsprechend benötigen die Prozessoren mit mehreren Registern auch mehr derartige Befehle.

6800 Der *6800*, der nur die beiden Akkumulatoren mit den Bezeichnungen **A** und **B** als 8-Bit-Register kennt, kommt mit zwei Datentransport-Befehlen mit Register-Adressierung aus. Es wird hier der Buchstabe «T» als Abkürzung für das Wort TRANSFER verwendet. Mit dem Befehl

$$\text{TAB} = (\text{A}) \rightarrow \text{B} \qquad \boxed{16}$$

wird der Inhalt des Akku A zum Akku B transportiert, während der Befehl

$$\text{TBA} = (\text{B}) \rightarrow \text{A} \qquad \boxed{17}$$

den Inhalt von **B** nach **A** transferiert. Der Hexadezimalcode für den **6800** ersten Befehl in Maschinensprache lautet **16** und für den zweiten **17.**

6502 Beim *6502* gibt es vier 8-Bit-Register mit den Bezeichnungen **A** = Akkumulator, **X** = Indexregister X, **Y** = Indexregister Y und **S** = Stapelzeiger. Im Befehlssatz werden nicht alle theoretisch möglichen Transferoperationen zugelassen. Es gibt aber immerhin schon 6 Transportbefehle mit Register-Adressierung.

$$\text{TAX} = (\text{A}) \rightarrow \text{X} \qquad \boxed{\text{AA}}$$
$$\text{TAY} = (\text{A}) \rightarrow \text{Y} \qquad \boxed{\text{AB}}$$
$$\text{TSX} = (\text{S}) \rightarrow \text{X} \qquad \boxed{\text{BA}}$$
$$\text{TXA} = (\text{X}) \rightarrow \text{A} \qquad \boxed{\text{8A}}$$
$$\text{TXS} = (\text{X}) \rightarrow \text{S} \qquad \boxed{\text{9A}}$$

6502
$$\text{TYA} = (\text{Y}) \rightarrow \text{A} \qquad \boxed{98}$$

2650 Der *2650* kennt zwei Bezeichnungen auch für die Transportbefehle mit Register-Adressierung, wobei immer nur das Register 0 als Daten-

quelle oder Datenziel bestimmt werden kann. Mit LOAD (Laden) werden Transporte in das Register 0 und mit STORE (Speichern) Transporte aus dem Register 0 bezeichnet. Ein direkter Datentransport zwischen den Vielzweckregistern 1 bis 3 ist nicht möglich.

Die Operationscodes der Befehle sind so aufgebaut, daß im Bit Nr. 0 und Nr. 1 immer das Register, das als Datenziel oder Datenquelle dient, nach folgender Codierung eingesetzt werden muß, die dem Binärwert der Registerbezeichnung entspricht:

Reigster 0 = **00**
Register 1 = **01**
Register 2 = **10**
Register 3 = **11**

Die beiden Transportbefehle mit Register-Adressierung lauten:

LODZ,R = RO ← (R)	0	0	0	0	0	0	R
STRZ,R = (RO) → R	1	1	0	0	0	0	R

Nach dem Kürzel der Assemblersprache ist nach einem Komma immer die Bezeichnung des Registers, das bei den Ladebefehlen als Datenquelle und bei den Speicherbefehlen als Datenziel dienen soll, einzusetzen.

Wenn die Codierung für die Register in die Bit-Positionen 0 und 1 des Operationscodes eingesetzt werden, ergeben sich die folgenden hexadezimalen Codierungen für die Befehle in Maschinensprache:

	R0	R1	R2	R3
LODZ	00	01	02	03
STRZ	–	C1	C2	C3

Da es sinnlos ist, den Inhalt von Register 0 nach Register 0 abzuspeichern, entfällt der betreffende Operationscode. Das gleiche gilt eigentlich auch für den entsprechenden Ladebefehl mit dem Hexadezimalcode 00, der aber trotzdem im Befehlssatz enthalten ist.

2650

Der *8080* hat insgesamt 7 CPU-Register mit der Bezeichnung **A** für den Akkumulator und **B, C, D, E, H** und **L** für die Hilfsregister. Der Datentransport ist zwischen allen Registern möglich, und so ergeben sich insgesamt 63 verschiedene Ein-Byte-Befehle mit Register-Adressierung. Die

8080

Kürzel der Befehle werden aus dem Wort MOVE (Bewegen) gebildet. Der Aufbau der Befehle ist immer wie folgt:

$$\text{MOV } R_{\text{Ziel}}, R_{\text{Quelle}} = R_Z \leftarrow (R_Q)$$

0	1	R_Z	R_Q

In der Assemblersprache des *8080* stehen das Ziel- und das Quellenregister durch ein Komma getrennt im Operandenfeld. Hierbei ist zu beachten, daß das Zielregister immer links vom Komma und das Quellenregister immer rechts davon steht. Die Transportrichtung ist also, bildlich dargestellt, immer von rechts nach links.

Der Code des Befehls ergibt sich, wenn in die Bitpositionen 0, 1 und 2 das Quellenregister und in Nr. 3, 4 und 5 das Zielregister entsprechend der folgenden Codierung eingesetzt wird:

Akkumulator = **111**

Register B = **000**	Register C = **001**
Register D = **010**	Register E = **011**
Register H = **100**	Register L = **101**

M (Indirekt HL) = **110**

Eine besondere Bedeutung hat die Codierung **110.** Hiermit wird nicht der Inhalt des Registerpaars HL transportiert, sondern der Inhalt der Adresse, die in dem Registerpaar HL steht. Dabei enthält das Register H den oberen Adreßteil (HIGH) und das Register L den unteren Adreßteil (LOW). Der Inhalt dieser Adresse wird mit «**M**» von MEMORY bezeichnet. MOVE-Befehle mit der Codierung **110** als Quelle oder Ziel erlauben also den Datenverkehr mit dem Speicher, ohne daß eine Adressenangabe im Operandenfeld des Befehls benötigt wird. Diesen Befehlsaufbau kennen die anderen Prozessoren nicht.

In hexadezimal codierter Maschinensprache können die Transportbefehle des *8080* mit Register-Adressierung in der folgenden Matrix dargestellt werden:

	Datenziel							
	A	B	C	D	E	H	L	M
A	–	47	4F	57	5F	67	6F	77
B	78	–	48	50	58	60	68	78
C	79	41	–	51	59	61	69	79
D	7A	42	4A	–	5A	62	6A	7A
E	7B	43	4B	53	–	63	6B	7B
H	7C	44	4C	54	5C	–	6C	7C
L	7D	45	4D	55	5D	65	–	7D
M	7E	46	4E	56	5E	66	6E	–

Datenquelle

Der *8080* kennt dann noch drei Tauch-(EXCHANGE-)Befehle, mit denen der Inhalt von Registern ausgetauscht wird.

SPHL = (HL) ↔ (SD) F9
XCHG = (HL) ↔ (DE) EB
XTHL = (HL) ↔ (PC) E9

8080

4.3.2 Zwei-Byte-Befehle mit unmittelbarer Adressierung

Bei allen Mikroprozessoren gibt es Befehle, bei denen ein im zweiten Befehls-Byte stehendes Bitmuster unmittelbar, englisch IMMIDIATE, in ein Register, manchmal auch in eine Speicherstelle, geladen werden kann. Das erste Befehls-Byte enthält immer den Operationscode, der das gewünschte Zielregister enthalten muß, während im zweiten Befehls-Byte eine beliebige, Information mit 8 Binärstellen steht. Dieses Bitmuster kann beliebig, z.B. als positive Zahl zwischen **0** und **255**, oder als Zahl im Zweier-Komplement zwischen **−128** und **+127** oder aber auch in anderer Weise interpretiert werden. Das zweite Befehls-Byte wird auch mit *Wert*, VALUE, *Konstante* oder *Maske* bezeichnet. Da es im allgemeinen als Befehlsbestandteil in einem ROM steht, kann es nicht im Programm geändert werden.

Beim *2650* und beim *8080* können alle CPU-Register unmittelbar mit einer Konstante geladen werden. Der Operationscode enthält also auch immer das Zielregister in codierter Form, wie im Abschnitt 4.3.1 bei den Befehlen mit Register-Adressierung beschrieben.

Beim *6502* können nur der Akkumulator und die Register X und Y und beim *6800* die beiden Akkumulatoren, das Indexregister und der Stapelzeiger unmittelbar geladen werden. Es ergeben sich somit die folgenden Befehle für die 4 Prozessoren:

2650

LODI,R Wert = R ← Datenbyte

0	0	0	0	0	I	R

Wert

	R0	R1	R2	R3
LODI	04	05	06	07

In der Assemblersprache des *2650* ist der Buchstabe **I** das Kennzeichen
2650 für einen Befehl mit unmittelbarer Adressierung.

6502

LDA	# Wert = A ← Datenbyte	A9	Wert
LDX	# Wert = X ← Datenbyte	A2	Wert
LDY	# Wert = Y ← Datenbyte	A0	Wert

Die Befehle mit unmittelbarer Adressierung werden durch das Zei-
6502 chen # vor dem Wert im Operandenfeld gekennzeichnet.

6800

LDAA	# Wert = A ← Datenbyte	86	Wert
LDAB	# Wert = B ← Datenbyte	C6	Wert
LDS	# Wert = SP ← Datenwort	8E	Datenwort
LDX	# Wert = X ← Datenwort	CE	Datenwort

Die Kennzeichnung als Befehle mit unmittelbarer Adressierung erfolgt in gleicher Weise wie beim *6502*. Da Indexregister und Stapelzeiger 16-Bit-Register sind, müssen sie auch mit 2 Byte geladen werden.

6800
8080

MVI R,Wert = R ← Datenbyte

0	0	R_Z	I	I	0

Wert

	A	B	C	D	E	H	L	M
MVI	3E	06	0E	16	1E	26	2E	36

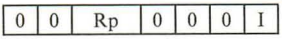
Eine besondere Kennzeichnung des Operanden bei unmittelbarer Adressierung ist nicht notwendig, da der mnemonische Kürzel **MVI** nur für diese Befehle verwendet wird.

Als Besonderheit kennt der *8080* noch einen Befehl, durch den ein Registerpaar unmittelbar geladen werden kann.

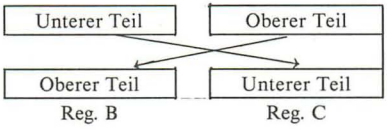
LXI R,Datenwort = Rp ← Datenwort

Da ein Registerpaar, das 16 Bit Information aufnehmen kann, geladen wird, handelt es sich um einen Drei-Byte-Befehl mit einer Codierung des Registerpaars im ersten Byte. Zu beachten ist, daß die Reihenfolge der beiden Befehls-Byte anders ist als die Abspeicherung in den Registern.

Befehl:



Inhalt Registerpaar:
z.B. B =



Ziel →	BC	DE	HL	SP
LXI	01	11	21	31

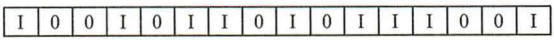
In der *8080*-Assemblersprache wird das Registerpaar immer nur mit dem 1. Register des Paars, also «B» für «BC», bezeichnet. **8080**

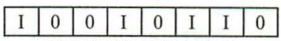
4.3.3 Drei-Byte-Befehle mit absoluter Adressierung

Zur vollständigen Darstellung einer Speicheradresse werden bei den meisten Mikroprozessoren 16 Bit, also zwei Byte, benötigt. Alle Prozessoren haben darum auch Drei-Byte-Befehle in ihrem Befehlssatz, bei denen das zweite und dritte Byte zur Darstellung der Operandenadresse verwendet wird, während das erste Befehls-Byte immer den Operationscode enthält.

Für diese Art der Adressierung haben leider die verschiedenen Mikroprozessor-Hersteller auch verschiedene Ausdrücke gewählt. Am meisten spricht man von absoluter Adressierung (englisch ABSOLUTE). Es werden aber auch die Ausdrücke EXTENDED und DIRECT verwendet.

Eine als sechzehnstellige Binärzahl geschriebene Adresse kann in zwei achtstellige Zahlen, also in zwei Byte, aufgeteilt werden.





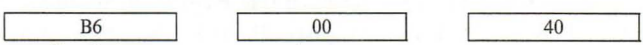
HIGH ORDER BYTE
MOST SIGNIFICANT BYTE
UPPER BYTE
(oberer Adreßteil)

LOW ORDER BYTE
LEAST SIGNIFICANT BYTE
LOWER BYTE
(unterer Adreßteil)

Im Deutschen bezeichnet man die Teile wohl am besten als *oberen* und *unteren Adreßteil,* während im Englischen eine ganze Reihe verschiedener Ausdrücke verwendet wird, die alle das gleiche bedeuten.

Leider sind sich nun die Entwickler der einzelnen Mikroprozessoren bezüglich der Reihenfolge, in der die einzelnen Adreßteile in das zweite und dritte Befehls-Byte zu schreiben sind, nicht einig geworden.

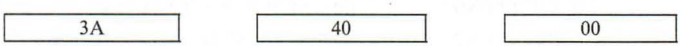
Beim *2650* und beim *6800* entspricht die Reihenfolge auch unserer normalen Schreibweise. Der obere Adreßteil steht im zweiten Byte, also links neben dem unteren Adreßteil.



Op.-Code | oberer | unterer

Adreßteil

Demgegenüber muß beim *6502* und beim *8080* genau umgekehrt programmiert werden. Im zweiten Befehls-Byte steht der untere Adreßteil, gefolgt vom dritten Befehls-Byte mit dem oberen Adreßteil.



Op.-Code | unterer | oberer

Adreßteil

Dies kann am Anfang zu Schwierigkeiten führen, wenn ein Programm in Assemblersprache mit hexadezimaler Befehlscodierung geschrieben wird. In der Assemblersprache wird die Adresse in normaler Schreibweise, d.h. also mit dem oberen Teil links und dem unteren Teil rechts, geschrieben, während man dann für die Codierung die beiden Adreßteile austauschen muß.

Im einzelnen können die besprochenen Prozessoren die folgenden Datentransfer-Befehle mit absoluter Adressierung ausführen:

6502

Beim *6502* kann der Akkumulator und die beiden Indexregister durch absolute Adressierung geladen bzw. ihr Inhalt abgespeichert werden. Die Befehle lauten wie folgt:

LDA	Adr. = (Adr.) → A	AD	Unt. Adr.	Ob. Adr.
LDX	Adr. = (Adr.) → X	AE	Unt. Adr.	Ob. Adr.
LDY	Adr. = (Adr.) → Y	AC	Unt. Adr.	Ob. Adr.
STA	Adr. = (A) → Adr.	8D	Unt. Adr.	Ob. Adr.
STX	Adr. = (X) → Adr.	8E	Unt. Adr.	Ob. Adr.
STY	Adr. = (Y) → Adr.	8C	Unt. Adr.	Ob. Adr.

6502

Beim *6800* können die beiden Akkumulatoren mit absoluter Adressierung, die in den Firmenbeschreibungen EXTENDED genannt wird, arbeiten. **6800**

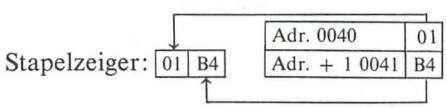
LDAA	Adr. = A ←(Adr.)	B6	Ob. Adr.	Unt. Adr.
LDAB	Adr. = B ←(Adr.)	F6	Ob. Adr.	Unt. Adr.
STAA	Adr. = (A) → Adr.	B7	Ob. Adr.	Unt. Adr.
STAB	Adr. = (B) → Adr.	F7	Ob. Adr.	Unt. Adr.

Darüber hinaus gibt es bei diesem Prozessor noch den Stapelzeiger (STACK POINTER) und das Indexregister, jeweils mit einer Kapazität von 16 Bit. Die Befehle für das Laden und Abspeichern dieser Register mit absoluter Adressierung sind zwar ähnlich aufgebaut, die Wirkung ist aber noch etwas anders. Da in einer Adresse immer nur 8 Bit unterzubringen sind, der Inhalt der Register jedoch 16 Bit beträgt, müssen immer zwei Speicherplätze beim Datentransport als Ziel oder Quelle dienen.

> Bei Datentransport-Befehlen für 16-Bit-Register des *6800* wird immer nur die Adresse des in der Reihenfolge niedrigeren Speicherplatzes genannt, dessen Inhalt bei den Ladebefehlen in den oberen Teil des Registers kommt, während der Inhalt der dann folgenden Adresse in den unteren Teil des Registers übernommen wird.

Ein Ladebefehl für den Stapelzeiger hat also folgende Wirkung:

LDS Adr. = SP ← (Adr.) (Adr. + 1)



Dementsprechend sind auch die anderen Lade- und Speicherbefehle für die 16-Bit-Register aufgebaut:

6800

LDS	Adr. = SP	← (Adr.) (Adr. + 1)	BE	Ob. Adr.	Unt. Adr.
LDX	Adr. = X	← (Adr.) (Adr. + 1)	FE	Ob. Adr.	Unt. Adr.
STS	Adr. = (SP)	→ Adr., Adr. + 1	BF	Ob. Adr.	Unt. Adr.
STX	Adr. = (X)	→ Adr., Adr. + 1	FF	Ob. Adr.	Unt. Adr.

8080

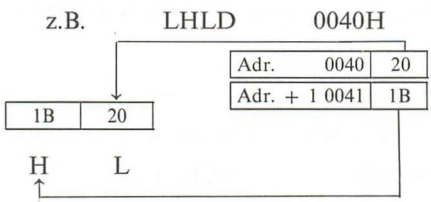
Obwohl der *8080* eine größere Anzahl Register besitzt, kann nur der Akkumulator direkt Daten aus den Speicherstellen mit absoluter Adressierung übernehmen bzw. dorthin abspeichern. In den Firmenschriften wird diese Adressierungsart mit DIRECT bezeichnet.

LDA	Adr. = A	← (Adr.)	3A	Unt. Adr.	Ob. Adr.
STA	Adr. = Adr.	← (A)	32	Unt. Adr.	Ob. Adr.

Als Besonderheit hat der *8080* aber noch die Möglichkeit, zwei Register zu einem Registerpaar zusammenzufassen. Im Befehlssatz gibt es darum auch noch zwei Befehle, durch die das Registerpaar HL absolut geladen bzw. sein Inhalt mit absoluter Adressierung abgespeichert werden kann.

LHLD	Adr. = H,L	← (Adr. + 1) (Adr.)	2A	Unt. Adr.	Ob. Adr.
SHLD	Adr. = Adr. + 1, Adr.	← (H,L)	22	Unt. Adr.	Ob. Adr.

Auch bei diesen Befehlen wird der Inhalt von zwei Speicherstellen benötigt, da es sich bei einem Registerpaar um ein Register mit 16 Bit Inhalt handelt. Beim Laden übernimmt das Register L den Inhalt der im Befehl stehenden Adresse und das Register H den Inhalt der nächstfolgenden Adresse.



Dies ist etwas verwirrend. Man muß aber bedenken, daß im Speicher immer zuerst der niederstwertige, also der untere Adreßteil, und dann im nächsten Byte der höchstwertige, also der obere Adreßteil, steht. Dem-

gegenüber enthält das Registerpaar HL, wenn es zum Abspeichern einer Adresse benutzt wird, im Register H immer den oberen (HIGH) und im Register L den unteren (LOW) Adreßteil.

Das gleiche gilt auch für den Speicherbefehl SHLD. Der Inhalt von L geht in den im Befehl adressierten Speicherplatz, der Inhalt von H in den nächstfolgenden.

8080

2650

Beim *2650* können alle erreichbaren CPU-Register mit absoluter Adressierung arbeiten. Die Operationscode der entsprechenden Lade- und Speicherbefehle haben also wieder im ersten Befehls-Byte ein Codefeld, in das das Ziel- bzw. Quellenregister einzusetzen ist.

Es gibt aber auch bei diesem Prozessor wieder eine Besonderheit, die man kennen muß, wenn man seine Befehle verstehen will.

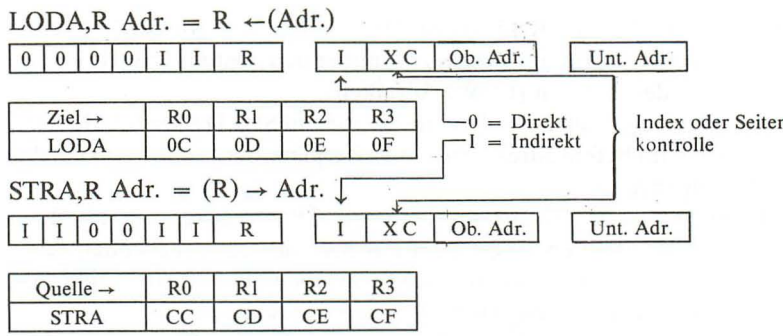
Der *2650* arbeitet mit einer 15-Bit-Adresse, kann also nur 32 768 verschiedene Adressen ansprechen. Dies braucht durchaus kein Nachteil zu sein, da es fast nie vorkommt, daß der volle Adreßraum in einem Mikrocomputer ausgenutzt wird.

Der Vorteil dieser Regelung besteht darin, daß ein Bit, und zwar das Bit Nr. 7, jeweils im zweiten Befehls-Byte noch für den Operationscode mit ausgenutzt werden kann, da ja die dann noch verbleibenden 15 Bit im zweiten und dritten Befehls-Byte für die Adressierung ausreichen.

Das Bit Nr. 7 im zweiten Befehls-Byte wird als *Indirekt-Bit* bezeichnet. Wenn dort eine **0** steht, arbeitet der Prozessor mit direkter Adressierung, steht dort eine **1**, ist indirekte Adressierung programmiert.

Aber auch Bit Nr. 5 und 6 haben beim *2650* noch eine besondere Bedeutung. Der gesamte zur Verfügung stehende Speicherraum ist nämlich in *4 Seiten* (PAGE) eingeteilt, wie dies auf Bild 4.3 dargestellt ist. Da der Inhalt einer Seite immer nur 8192 Speicherplätze umfaßt, kann eine Adresse auf einer bestimmten Seite durch maximal 13 Bit angesprochen werden. Man geht nun beim *2650* davon aus, daß durch Datentransport-Befehle mit absoluter Adressierung immer nur eine Adresse in der Seite, die gerade vom Programm abgearbeitet wird, angesprochen werden kann. So steht also das Bit Nr. 5 und 6 im zweiten Befehls-Byte wiederum für einen Teil des Operationscodes zur Verfügung. Es können dort verschiedene Arten der Index-Adressierung programmiert werden, worauf im Abschnitt 4.3.7 noch näher eingegangen wird. Für Sprungbefehle steht der volle Speicherraum zur Verfügung.

Die Datentransport-Befehle mit absoluter Adressierung haben beim *2650* also folgenden Aufbau:



Seite PAGE	PAGE CONTROL	1. Adresse der Seite Direkt	1. Adresse der Seite Indirekt	letzte Adresse der Seite Direkt	letzte Adresse der Seite Indirekt	Speicherbereich Dezimal
0	0 0	0000	8000	1FFF	9FFF	0 ... 8191
1	0 1	2000	A000	3FFF	BFFF	8192 ... 16383
2	1 0	4000	C000	5FFF	DFFF	16384 ... 24575
3	1 1	6000	E000	7FFF	FFFF	24576 ... 32767

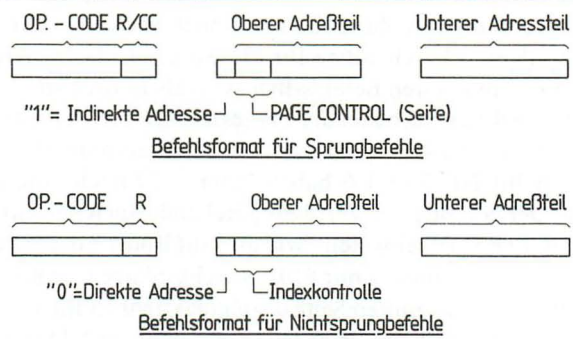


Bild 4.3 Speicherorganisation und Befehlsformate für Drei-Byte-Befehle beim 2650

Beim *2650* werden die Bit Nr. 5, 6 und 7 des zweiten Befehlsbyte mit für den Operationscode benutzt. Drei-Byte-Befehle mit absoluter Adressierung sind darum unterschiedlich aufgebaut, je nachdem, ob es sich um Sprungbefehle oder um andere Befehle handelt.

Der gesamte adressierbare Speicherraum mit den Adressen hexadezimal **0000** bis **7FFF** ist in vier Seiten (PAGES) eingeteilt, die entsprechend den höchstwertigsten Adreßbit mit **00, 01, 10** und **11** gekennzeichnet werden. Bei Sprungbefehlen, bei denen die Überschreitung der Seitengrenzen möglich ist, kommt die Seitenbezeichnung als PAGE CONTROL in die Bit Nr. 5 und 6 des zweiten Befehlsbyte.

Bei allen anderen Befehlen ist die Überschreitung des Seitenbereichs nicht möglich, da hier die Bit Nr. 5 und 6 für die Indexkontrolle verwendet werden (siehe auch Abschnitt 4.3.7).

4.3.4 Zwei-Byte-Befehle mit Seite-Null-Adressierung

In vielen Fällen bezeichnet man den Teil des Speicherbereichs, der allein durch eine 8-Bit-Adresse zu adressieren ist, als Seite Null. Da in 8 Bit 256 verschiedene Bitmuster unterzubringen sind, können auf diese Weise auch 256 Adressen mit nur zwei Byte langen Befehlen, von denen das erste Befehls-Byte den Operationscode und das zweite die Adresse auf Seite Null enthält, angesprochen werden. In vielen Fällen wird dieser Teil des Speicherbereichs durch einen RAM-Speicher besetzt, und man kann dort Daten zwischenspeichern und wieder abrufen und im Programm verwenden.

Von den hier besprochenen Prozessoren kennen allerdings nur die Typen *6502* und *6800* diese Art der Adressierung bei den Datentransport-Befehlen. Beim *2650* und auch *8080* werden Teile der Seiten Null oft als Einsprungadressen für Unterprogramme verwendet. In diesem Fall wird in den Bereich der Seite Null des Adressierraums dann ein ROM-Speicherbaustein angeschlossen.

Beim *6502* stehen für das Laden und Abspeichern der drei 8-Bit-CPU-Register die folgenden Befehle zur Verfügung. **6502**

LDA Adr.	= (Adr.) → A	A5	Adr.
LDX Adr.	= (Adr.) → X	A6	Adr.
LDY Adr.	= (Adr.) → Y	A4	Adr.
STA Adr.	= (A) → Adr.	85	Adr.
STX Adr.	= (X) → Adr.	86	Adr.
STY Adr.	= (Y) → Adr.	84	Adr.

6502

Beim *6800* muß wieder entschieden werden, ob 8-Bit-Register oder 16-Bit-Register geladen bzw. abgespeichert werden sollen. In der Sprache des Herstellers wird bei diesem Prozessor die Seite-Null-Adressierung als DIRECT bezeichnet. Die Befehle für das Laden und Abspeichern der Inhalte der beiden Akkumulatoren lauten wie folgt: **6800**

LDAA Adr.	= A ← (Adr.)	96	Adr.
LDAB Adr.	= B ← (Adr.)	D6	Adr.
STAA Adr.	= (A) → Adr.	97	Adr.
STAB Adr.	= (B) → Adr.	D7	Adr.

Bei den beiden 16-Bit-Registern werden wieder zwei Speicherstellen als Datenquelle oder Datenziel benötigt. Ähnlich wie bei der absoluten

Adressierung wird der Inhalt der im Befehl angesprochenen Adresse in den oberen Teil des Registers und der Inhalt der nächstfolgenden Adresse in den unteren Teil des Registers geladen. Es gibt also die folgenden vier Befehle für das Laden bzw. Abspeichern des Stapelzeiger- und Indexregisterinhalts.

6800

LDX Adr. = X	←(Adr.) (Adr. + 1)	DE	Adr.
LDS Adr. = SP	←(Adr.) (Adr. + 1)	9E	Adr.
STX Adr. = (X)	→ Adr., Adr. + 1	DF	Adr.
STS Adr. = (SP)	→ Adr., Adr. + 1	9F	Adr.

4.3.5 Zwei-Byte-Befehle mit relativer Adressierung

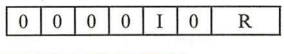
Wie schon erklärt wurde, wird bei relativer Adressierung die effektive Adresse, d.h. die Adresse, in der der Operand zu finden ist, erst durch eine Rechnung ermittelt. Hierbei wird der Binärwert des Bitmusters im zweiten Befehls-Byte als Zahl im Zweier-Komplementsystem gewertet und zur Binärzahl, die sich nach Ausführung des letzten Befehls im Befehlsadreß-Register befindet, addiert, wie dies im Abschnitt 3.8.6 erklärt wurde.

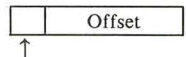
2650 Von den hier besprochenen Prozessoren kennt nur der *2650* auch relative Adressierung bei Datentransportbefehlen. Es gibt aber auch wieder eine Besonderheit.

Da das Bit Nr. 7 im zweiten Befehls-Byte benutzt wird, um auch bei Befehlen mit relativer Adressierung direkte oder indirekte Adressenfindung zu spezifizieren, stehen für die Darstellung der relativen Verschiebung nur die Bit-Nr. 0 bis 6, d.h. insgesamt sieben Bit, zur Verfügung. In diesen sieben Bitpositionen lassen sich Dezimalzahlen zwischen **+63** und **−64** unterbringen, so daß hierdurch die höchstmögliche *Verschiebung* (OFFSET) bei relativer Adressierung bestimmt wird.

Die Datentransport-Befehle mir relativer Adressierung sind im einzelnen wie folgt aufgebaut:

LODR,R Offset = R ←(PC.+Of.)

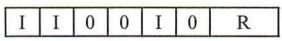


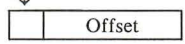


Ziel →	R0	R1	R2	R3
LODR	08	09	0A	0B

0 = Direkt
I = Indirekt

STRR,R Offset = (R) → PC.+Of.





Quelle	R0	R1	R2	R3
STRR	C8	C9	CA	CB

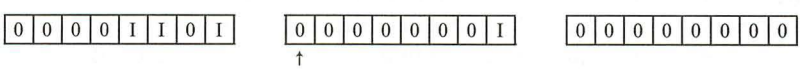
4.3.6 Befehle mit indirekter Adressierung **2650**

Von den hier besprochenen Mikroprozessoren kennt nur der *2650* die in Abschnitt 3.8.7 besprochene indirekte Adressierung bei Datentransport-Befehlen. Abweichend von dem sonst üblichen Befehlsaufbau wird beim *2650* jedoch die indirekte Adressierung nicht schon im Operationscode des ersten Befehls-Bytes spezifiziert, sondern es wird das Bit Nr. 7 im zweiten Befehls-Byte zur Kennzeichnung einer indirekten Adresse verwendet. Für alle Befehle mit absoluter oder relativer Adressierung kann also noch zusätzlich bestimmt werden, daß die aus dem Befehl hervorgehende Adresse nicht direkt als Operandenadresse, sondern als indirekte Adresse anzusehen ist.

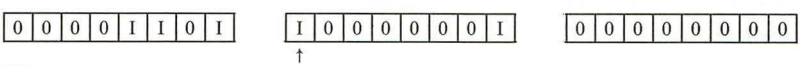
Wird das Bit Nr. 7 im zweiten Befehls-Bit auf **1**, also auf «indirekt», gesetzt, so bedeutet dies, daß der Prozessor den Inhalt der im Befehl angezogenen Adresse und den Inhalt der nächstfolgenden Adresse als weitere Adresse erkennt, die erst als effektive Operandenadresse gilt.

Befehle mit direkter absoluter Adressierung und mit indirekter absoluter Adressierung haben beim *2650* den gleichen Operationscode, sie unterscheiden sich also im ersten Befehls-Byte nicht. Im nachfolgenden Beispiel wird dies an Ladebefehlen in das Register 1 dargestellt.

LODA,1 H'0100' 0D 01 00



LODA,1 *H'0100' 0D 81 00



Im ersten Befehl wird der Inhalt der Speicherstelle mit der hexadezimalen Adresse **0100** direkt in das Register 1 geladen, während beim zweiten Befehl die Adresse **0100** lediglich als indirekte Adresse gilt. Man spricht auch davon, daß in diesem Fall die Adresse **0100** und die nächstfolgende Adresse **0101** als *Zeiger* (POINTER) verwendet wird, weil der Inhalt dieser Adressen auf die eigentliche Operandenadresse zeigt.

In der Assemblersprache des *2650* bedeutet ein vorangestellter Stern (*), daß der nachfolgende Ausdruck als indirekte Adresse zu behandeln ist. Bei der hexadezimalen Codierung in der Maschinensprache muß man bei indirekter Adressierung zu der hexadezimal ausgedrückten Adresse noch den Wert hex **8000** hinzuzählen und das Ergebnis dann in das zweite und dritte Befehls-Byte einsetzen.

Unter Beachtung der gleichen Regeln kann auch bei den Speicherbefehlen zusätzlich indirekte Adressierung angezogen werden. Auch bei allen Befehlen mit relativer Adressierung ist zusätzlich indirekte Adressenangabe möglich, da für die Darstellung der relativen Verschiebung nur die Bit-Nr. 0 bis 6 im zweiten Befehls-Byte verwendet werden. Das Bit Nr. 7 dient also auch hier zur Anweisung, daß es sich um eine indirekte Adresse handelt, wenn es auf «1» gesetzt ist. So lauten z.B. die Befehle für das Abspeichern des Inhalts des Registers 3 **9** Speicherplätze vorwärts bei relativer direkter Adressierung:

STRR,3 09 | CB | 09 |

bei relativer indirekter Adressierung:

2650 STRR,3 *09 | CB | 89 |

8080 Die indirekte Adressierung beim *8080* arbeitet ganz anders und entspricht eigentlich der im Abschnitt 3.8.2 beschriebenen Adressierung durch ein Registerpaar. Bei diesem Prozessor gibt es eine Reihe von Befehlen, bei denen im Befehl überhaupt keine Operandenadresse steht. Statt dessen wird das in einem Registerpaar stehende Bitmuster, das in diesem Fall eine Breite von 16 Bit hat, als Operandenadresse interpretiert. Einige dieser Befehle wurden schon im Abschnitt 4.3.1 und 4.3.2 vorgestellt. Es handelt sich um die Befehle, bei denen im Operationscode als Registercodierung der Wert **110** enthalten ist. Bei diesen Befehlen wird der Inhalt des Registerpaares HL als Adresse interpretiert, in der sich der Operand befindet.

Darüber hinaus gibt es bei diesem Prozessor noch je zwei weitere Ein-

Byte-Befehle, mit denen jeweils ein Registerpaar als Adreßzeiger für das indirekte Laden des Akkumulators bzw. das indirekte Abspeichern des Akkumulatorinhalts verwendet werden kann.

LDAX	B = A ← ((BC))	0A
LDAX	D = A ← ((DE))	1A
STAX	B = ((BC)) ← (A)	02
STAX	D = ((DE)) ← (A)	12

Die beschriebene Art der indirekten Adressierung durch ein Registerpaar kann in ihrer Wirkung auch die sonst bei den meisten anderen Mikroprozessoren übliche Index-Adressierung, die im nächsten Abschnitt besprochen wird, ersetzen. **8080**

4.3.7 Befehle mit Index-Adressierung

Die Mikroprozessoren *2650, 6502* und *6800* können auch bei Datentransport-Befehlen mit Index-Adressierung, die man auch indizierte Adressierung nennt, wie im Abschnitt 3.8.8 beschrieben, arbeiten. Das Prinzip der Index-Adressierung besteht darin, daß die gültige Operandenadresse immer aus der Summe von zwei Binärzahlen errechnet wird. Eine davon steht im Befehl, die zweite im Indexregister des Prozessors. Trotzdem gibt es bei den hier besprochenen Prozessortypen auch wieder Unterschiede.

Beim *6502* gibt es zwei 8-Bit-Indexregister, die Register X und Register Y genannt werden. Bei vielen Befehlen kann eines dieser Register dazu benutzt werden, die im Befehl stehende Adresse noch zu verändern. Dies ist vor allem bei Programmschleifen wichtig, wie noch im einzelnen im Kapitel 5 gezeigt werden wird. **6502**

Beim *6502* kann Index-Adressierung zusätzlich sowohl bei Befehlen mit absoluter Adressierung als auch bei Befehlen mit Seite-Null-Adressierung angewandt werden. Es sind allerdings nicht alle denkbaren Kombinationen möglich. In der Assemblersprache wird Index-Adressierung im Operandenfeld durch ein X oder Y nach einem Komma gekennzeichnet. Die mnemonischen Befehlskürzel sind bei diesem Prozessor für alle Adressierarten gleich. Unterschiedlich ist dagegen der Operationscode im ersten Befehls-Byte, aus dem hervorgeht, mit welcher Adressierart gearbeitet werden soll.

Es gibt folgende Zwei-Byte-Datentransport-Befehle für den Adreßraum auf Seite Null mit zusätzlicher Indexierung:

LDA	Adr., X = A ←(Adr. + X)	B5 Adr.
LDX	Adr., Y = X ←(Adr. + Y)	B6 Adr.
LDY	Adr., X = Y ←(Adr. + X)	B4 Adr.
STA	Adr., X = (A) → Adr. + X	95 Adr.
STX	Adr., Y = (X) → Adr. + Y	96 Adr.
STY	Adr., X = (Y) → Adr. + X	94 Adr.

Ebenso gibt es auch Drei-Byte-Datentransport-Befehle mit vollständiger Adresse im zweiten und dritten Byte und zusätzlicher Indizierung durch das Register X oder Register Y.

LDA	Adr., X = A ←(Adr. + X)	BD Unt. Adr. Ob. Adr.
LDA	Adr., Y = A ←(Adr. + Y)	B9 Unt. Adr. Ob. Adr.
LDX	Adr., Y = X ←(Adr. + Y)	BE Unt. Adr. Ob. Adr.
LDY	Adr., X = Y ←(Adr. + X)	BC Unt. Adr. Ob. Adr.
STA	Adr., X = (A) → Adr. + X	9D Unt. Adr. Ob. Adr.
STA	Adr., Y = (A) → Adr. + Y	99 Unt. Adr. Ob. Adr.

6502

6800 Auch der *6800* kennt eine große Anzahl von Befehlen mit Index-Adressierung. Da bei diesem Prozessor das Indexregister eine Kapazität von 16 Bit besitzt, handelt es sich bei allen Befehlen mit Index-Adressierung um Zwei-Byte-Befehle, bei denen das zweite Byte einen Wert enthält, der grundsätzlich zum Inhalt des Indexregisters hinzuaddiert wird, um die effektive Adresse, in der der Prozessor den Operanden zu suchen hat, zu bestimmen.

Es können sowohl die Inhalte der beiden Akkumulatoren A und B als auch die Inhalte des Indexregisters und des Stapelzeigers mit indizierter Adressierung geladen oder abgespeichert werden. Dabei ist zu bedenken, daß die beiden Akkumulatoren nur einen Inhalt von 8 Bit haben, also auch nur eine Speicherstelle benötigen, während bei den Befehlen, die das Indexregister und den Stapelzeiger (STACK POINTER) betreffen, immer zwei hintereinanderliegende Speicherstellen betroffen sind. Wie bei diesem Prozessor allgemein üblich, kommt der Inhalt der ersten Speicherstelle in den oberen Teil des Speichers und der Inhalt der nächstfolgenden Speicherstelle in den unteren Teil.

Die Datentransport-Befehle mit indizierter Adressierung sind beim *6800* wie folgt aufgebaut:

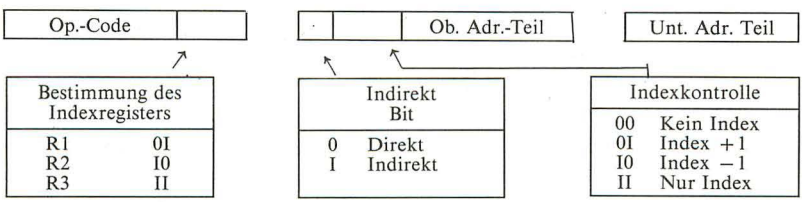
LDAA	Wert, X = (X + Wert) → A	6A	Wert
LDAB	Wert, X = (X + Wert) → B	6E	Wert
LDX	Wert, X = (X + Wert, X + Wert + 1) → X	EE	Wert
LDS	Wert, X = (X + Wert, X + Wert + 1) → SP	AE	Wert
STAA	Wert, X = (A) → X + Wert	A7	Wert
STAB	Wert, X = (B) → X + Wert	E7	Wert
STX	Wert, X = (X) → X + Wert, X + Wert + 1	EF	Wert
STS	Wert, X = (SP) → X + Wert, X + Wert + 1	AF	Wert

Für alle Befehle mit Index-Adressierung gilt beim *6800* folgende Rechnung

Basisadresse (Inhalt des Indexregisters)
+ Verschiebung (Wert im 2. Befehls-Byte)
ergibt effektive Adresse

Auch beim *6800* sind die Kürzel der Assemblersprache für alle Adressierarten gleich. Das Zeichen für Index-Adressierung ist der Buchstabe X nach einem Komma im Operandenfeld des Befehls. **6800**

Sehr vielseitig sind die Möglichkeiten der Index-Adressierung beim *2650*, da hier, wie schon früher angedeutet, bei allen Drei-Byte-Befehlen mit absoluter Adressierung, die keine Sprungbefehle sind, die Bit-Nr. 5 und 6 im zweiten Befehls-Byte für die Indexkontrolle nach folgendem Schema verwendet werden. **2650**



Man kann also in allen derartigen Befehlen bestimmen, ob ohne Index, nur mit Index oder mit Index und automatischer Erhöhung des Indexregisters oder mit Index und automatischer Erniedrigung des Indexregisters gearbeitet werden soll.

Wahlweise können beim *2650* die Register 1, 2 oder 3 als Indexregister spezifiziert werden. Grundsätzlich dient das Register 0 bei allen Befehlen mit Index-Adressierung als Ziel- oder Quellenregister bzw. als 1. Operandenadresse.

Wenn selbsttätige Erhöhung oder Erniedrigung des Indexregisters (Incrementierung oder Decrementierung) gefordert werden, so wird dieser Rechenvorgang **vor** der Berechnung der effektiven Operandenadresse durchgeführt.

Die Befehle mit Index-Adressierung haben beim *2650* den gleichen Operationscode wie Befehle ohne Index. Sie sind grundsätzlich wie folgt aufgebaut:

LODA,O Adr ,XR = RO ← (Adr. + XR)
LODA,O Adr ,XR+ = RO ← (Adr. + XR + 1)
LODA,O Adr ,XR− = RO ← (Adr. + XR −1)

Indexreg.	R1	R2	R3
LODA	09	0A	0B

1. Befehls-Byte

Oberer Teil der Seitenadresse		
Nur Ind.	Ind. +1	Ind. −1
+H'60'	+H'20	+H'40'

2. Befehls-Byte

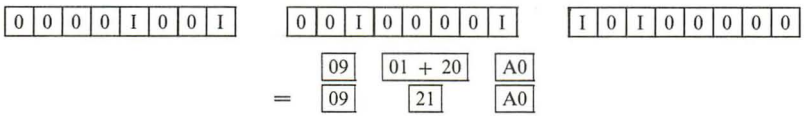
Unterer Teil der Adresse

3. Befehlsbyte

Beim Codieren von Befehlen mit Index-Registrierung muß man darauf achten, daß in den Operationscode im ersten Befehls-Byte der Code für das Indexregister einzusetzen ist. Im Assemblerausdruck steht das Indexregister im Argumentfeld nach dem Komma, während im Operandenfeld das Register 0 nach dem Komma stehen muß, da dieses ja als Operandenregister dient. Wenn automatische Erhöhung oder Erniedrigung gewünscht wird, muß in der Assemblersprache nach der Benennung des Indexregisters noch ein Plus- oder ein Minuszeichen angefügt werden.

In dem folgenden Beispiel ist ein Befehl dargestellt, mit dem das Register 0, indiziert durch Register 1, mit dem Inhalt der Speicherstelle **01A0** geladen wird.

LODA,O H'01A0',1+ = R0 ← (01A0 + (R1) +1)



Hierbei ist zu beachten, daß die erste mögliche Speicherstelle, auf die mit einem derartigen Befehl zugegriffen werden kann, die Speicherstelle **01A1** ist. Selbst wenn das Indexregister R1 vor dem Befehl auf **0** gestellt wurde, wird es durch den Befehl vor der Berechnung der effektiven Adresse um **1** erhöht, so daß durch diesen Befehl die Adresse **01A1** als effektive Adresse angesprochen wird. **2650**

4.3.8 Kombinationen von indirekter und Index-Adressierung

Etwas schwierig zu verstehen ist die beim *2650* und *6502* noch mögliche Kombination von indirekter und Index-Adressierung. Denn es gibt hier wieder zwei Möglichkeiten, die man *indirekte Adressierung mit nachheriger Indexierung* bzw. mit *vorheriger Indexierung* bezeichnet.

Die beiden Methoden dieser kombinierten Adressierung sind im Bild 4.4 dargestellt. Es ist zu erkennen, daß bei der indirekten Adressierung mit vorheriger Indexierung vom Prozessor bereits die indirekte Adresse, d.h. also die Zeigeradresse, durch Addition der im Befehl stehenden Adresse und des Inhalts des Indexregisters berechnet wird. In der so gefundenen Adresse steht dann wieder ein Teil der eigentlichen Operandenadresse, und die nächstfolgende Adresse enthält dann den zweiten Teil.

Bei der indirekten Adressierung mit nachheriger Indexierung steht im Befehl eine Adresse, die bereits ein Teil der Zeigeradresse enthält. Im nächstfolgenden Speicherplatz befindet sich dann der zweite Teil der Zeigeradresse. Die dort gefundene Adresse wird aber nun noch um den Inhalt des Indexregisters erhöht, und erst die Summe aus indirekter Adresse und Inhalt des Indexregisters ergibt die eigentliche Operandenadresse.

Beim *2650* ist nur indirekte Adressierung mit nachheriger Indexierung möglich, die allerdings bei allen Drei-Byte-Befehlen, die keine Sprungbefehle sind, spezifiziert werden kann. Ein derartiger Befehl ist dann z.B. wie folgt aufgebaut: **2650**

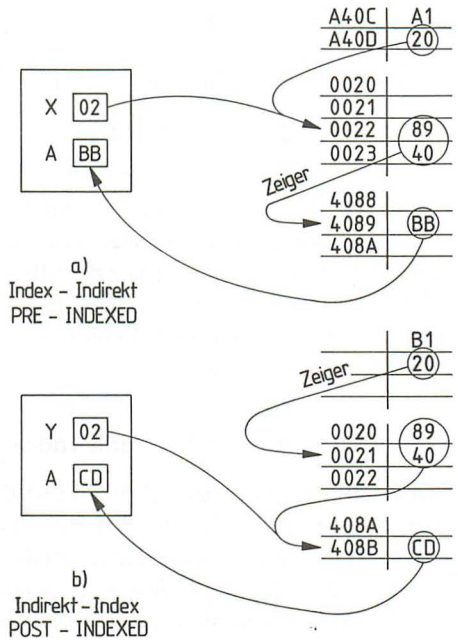
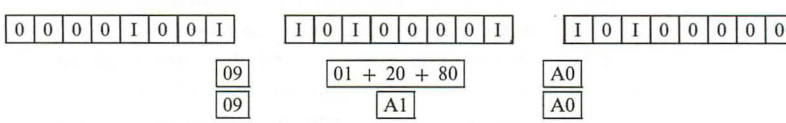


Bild 4.4 Kombination von Index- und indirekter Adressierung
Bei einigen Prozessoren ist es möglich, zunächst mit Hilfe des Indexregisters einen Zeiger zu bestimmen, der dann erst auf die Operanden-Adresse zeigt (a) (vorherige Indexierung = PRE-INDEXING). Bei anderen Typen steht im Befehl die Adresse des Zeigers, zu dessen Inhalt der Inhalt des Indexregisters noch hinzuaddiert werden muß, um die effektive Operandenadresse zu erhalten (b) nachherige Indexierung = POST-INDEXING).

LODA,0 *H'01A0,1+ = R0 ← ((1A0+(R1)+1))



Im Bit 7 des zweiten Befehls-Byte muß eine **1** gesetzt werden, und die Bit-Nr. 5 und 6 nehmen die Indexkontrollbit gemäß der im vorigen Abschnitt erklärten Tabelle auf. In der Assemblersprache muß das Argumentfeld den Stern für indirekte Adressierung und nach dem Komma das gewählte Indexregister enthalten.

2650

Beim *6502* sind Kombinationen von Index- und indirekter Adressierung nur für den Speicherraum auf Seite Null möglich. Das Indexregister X wird für indirekte Adressierung mit vorheriger Indexierung und das Indexregister Y für indirekte Adressierung mit nachheriger Indexierung verwendet. Beide Adressierarten können für das Laden des Akkumulators und für das Abspeichern seines Inhalts in eine Speicherstelle der Seite Null verwendet werden. Es sind dafür die folgenden Befehle vorgesehen: **6502**

LDA (Adr., X) = ((Adr. + (X)) → A	A1 Adr.
LDA (Adr.), Y = ((Adr.)) + (Y) → A	B1 Adr.
STA (Adr., X) = (A) → ((Adr. + (X))	81 Adr.
STA (Adr.), Y = (A) → ((Adr.)) + (Y)	91 Adr.

Während die Kürzel für den Operationscode in der Assemblersprache für alle gleichartigen Befehle auch für unterschiedliche Adressierarten gleichlautend sind, ist der Maschinencode natürlich für die verschiedenen Adressierarten unterschiedlich. Es handelt sich immer um Zwei-Byte-Befehle, so daß also nur der Adreßraum auf Seite Null mit diesen Befehlen angesprochen werden kann. **6502**

4.4 Datenverarbeitungs-Befehle

Nachdem die Datentransport-Befehle sehr ausführlich besprochen wurden, kann es jetzt bei den Datenverarbeitungs-Befehlen etwas schneller vorangehen, obwohl diese der Zahl nach in den Befehlssätzen einen großen Raum einnehmen.

Zunächst ist zu sagen, daß alle Mikrorpozessoren, die hier besprochen werden, fünf Möglichkeiten für die Verarbeitung, d.h. für die Verknüpfung von je zwei 8 Bit breiten Informationen besitzen. Es sind dies:

- Addition und Subtraktion im Zweier-Komplement, wie im Abschnitt 3.6.1 beschrieben, und
- die drei logischen Operationen UND, ODER sowie EXKLUSIV-ODER, die in 3.6.2 beschrieben wurden.

Bei den besprochenen Prozessoren sind bei den Datenverarbeitungs-Befehlen mit zwei Operanden im wesentlichen die gleichen Adressierarten möglich, wie sie für die Datentransport-Befehle beschrieben wurden.

Die Befehle der einzelnen Prozessoren werden darum im folgenden tabellarisch zusammengefaßt und Einzelheiten nur noch dann erklärt, wenn es Abweichungen von dem üblichen Schema gibt.

Recht unterschiedlich aufgebaut sind die Datenverarbeitungsbefehle, die nur einen Operanden betreffen, bei den verschiedenen Mikroprozessor-Typen. Bei ihnen gibt es aber meist nur wenige Adressierarten, so daß sie auch ohne große Schwierigkeiten verständlich sind.

Bei allen Datenverarbeitungs-Befehlen werden einzelne Bit der Programm-Status-Register beeinflußt. Es gelten hier die Regeln, die im Abschnitt 3.5 erklärt wurden. Geringfügige Abweichungen sind aus den Tabellen zu ersehen.

4.4.1 Befehle für die Verknüpfung von zwei Operanden

2650 Beim *2650* gibt es für jede der fünf möglichen Verknüpfungen vier Adressierarten und zusätzlich für relative und absolute Adressierung die indirekte und die Index-Adressierung. Assemblerausdruck, Operationscode in hexadezimaler Form für das erste Befehls-Byte, Veränderung des Programm-Status-Wortes und Wirkung des Befehls sind in der Tabelle in Bild 4.5 dargestellt.

Bei allen Befehlen mit Register-Adressierung enthält das Register 0 den ersten Operanden und nimmt später das Ergebnis der Operation auf. Das danach unter Zwischenschaltung eines Kommas im Assemblerausdruck genannte Register enthält dann den zweiten Operanden und wird durch den Befehl in seinem Inhalt nicht verändert.

Bei allen anderen Befehlen enthält das nach dem Komma im Assemblerausdruck genannte Register den ersten Operanden und nimmt auch wieder das Ergebnis auf. Der zweite Operand ist in der effektiven Adresse zu finden, die nach den Regeln der Adressierung mit Hilfe des Operanden im zweiten bzw. zweiten und dritten Byte des Befehls vom Prozessor bestimmt wird.

Eine Besonderheit beim *2650* besteht nun darin, daß durch Setzen oder Löschen gewisser Bit im Programm-Status-Register die Arbeitsweise des Prozessors noch beeinflußt werden kann. Der Prozessor hat zwei 8-Bit-Register, die als *oberes und unteres Programm-Status-Register* bezeichnet werden. In diesem Zusammenhang interessiert nur das untere Programm-Status-Register, das mit PSL (PROGRAM STATUS LOWER) bezeichnet wird. Wird durch einen Programmbefehl in diesem Register

BYTES	Assembler-Ausdruck	Operandenregister R0	R1	R2	R3	PS Wort Unteres CC	IDC	OVF	C	Befehlsablauf
1	ADDZ,R	80	81	82	83	●	●	●	●	(R0) + (R) → R0
2	ADDI,R Wert	84	85	86	87	●	●	●	●	(R) + Wert → R
2	ADDR R Offset	88	89	8A	8B	●	●	●	●	(R) + (PC+Of) → R
3	ADDA,R Adr.	8C	8D	8E	8F	●	●	●	●	(R) + (Adr.) → R
1	SUBZ,R	A0	A1	A2	A3	●	●	●	●	(R0) - (R) → R0
2	SUBI,R Wert	A4	A5	A6	A7	●	●	●	●	(R) - Wert → R
2	SUBR,R Offset	A8	A9	AA	AB	●	●	●	●	(R) - (PC+Of) → R
3	SUBA,R Adr.	AC	AD	AE	AF	●	●	●	●	(R) - (Adr.) → R
1	ANDZ,R	-	41	42	43	●				(R0) ∧ (R) → R0
2	ANDI,R Wert	44	45	46	47	●				(R) ∧ Wert → R
2	ANDR,R Offset	48	49	4A	4B	●				(R) ∧ (PC+Of) → R
3	ANDA,R Adr.	4C	4D	4E	4F	●				(R) ∧ (Adr.) → R
1	IORZ,R	60	61	62	63	●				(R0) ∨ (R) → R
2	IORI,R Wert	64	65	66	67	●				(R) ∨ Wert → R
2	IORR,R Offset	68	69	6A	6B	●				(R) ∨ (PC+Of) → R
3	IORA,R Adr.	6C	6D	6E	6F	●				(R) ∨ (Adr.) → R
1	EORZ,R	20	21	22	23	●				(R0) ⊻ (R) → R
2	EORI,R Wert	24	25	26	27	●				(R) ⊻ Wert → R
2	EORR,R Offset	28	29	2A	2B	●				(R) ⊻ (PC+Of) → R
3	EORA,R Adr.	2C	2D	2E	2F	●				(R) ⊻ (Adr.) → R

Condition Code
Interdigit Carry
Overflow
Carry

Bei Subtraktionen wird das Carry-Bit auf 1 gesetzt, wenn nicht geborgt wurde.

Unteres Programm - Status-Wort

CC1	CC2	IDC	RS	WC	OVF	COM	C	PSL
7	6	5	4	3	2	1	0	

Condition Code		CC1	CC2
Ergebnis:	positiv	0	1
	Null	0	0
	negativ	1	0

Anweisung:
1 = Mit Übertrag
0 = Ohne Übertrag

Bild 4.5 Operationscode der Datenverarbeitungs-Befehle beim 2650

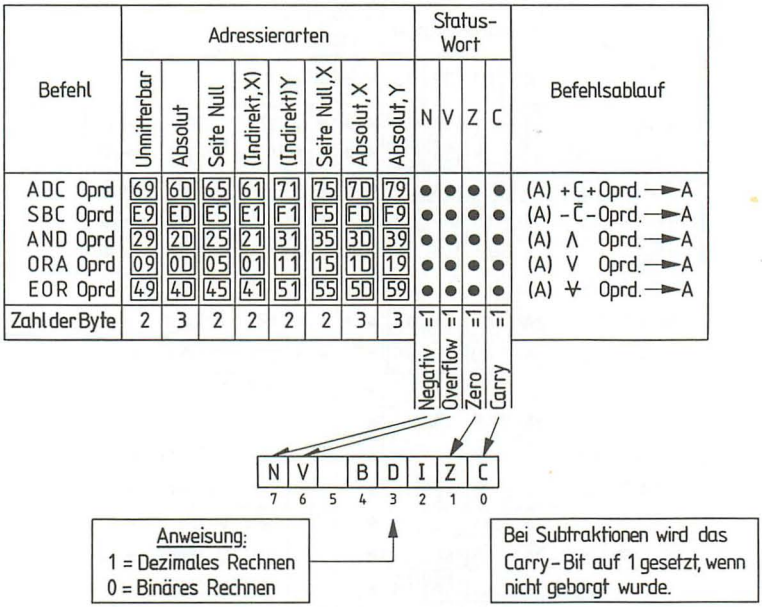


Bild 4.6 Datenverarbeitungs-Befehle beim 6502

das Bit Nr. 3, das in hexadezimaler Schreibweise mit **08** bezeichnet wird, auf **1** gesetzt, so arbeitet der Prozessor grundsätzlich mit CARRY, d.h. unter Beachtung des Übertrags-Bit. Wird an diese Stelle eine **0** gesetzt, so wird das Übertrags-Bit vom Prozessor nicht beachtet.

2650
6502

Beim *6502* gibt es fünf Grundbefehle mit jeweils acht verschiedenen Adressierarten für die Datenverarbeitung mit zwei Operanden (Bild 4.6). Bei den Additions- und Subtraktionsbefehlen wird das Carry-Bit grundsätzlich mitverarbeitet. Je nach dem Ergebnis der Datenverarbeitung werden die Ergebnis-Bit im Programm-Status-Wort überall dort gesetzt oder gelöscht, wo in der Tabelle ein Punkt steht.

Als Besonderheit gibt es beim 6502 noch die Möglichkeit, Additionen und Subtraktionen entweder im binären Zahlensystem oder dezimal im BCD-Zahlensystem auszuführen. Wenn durch einen entsprechenden Programmbefehl das Bit Nr. 3 im Programm-Status-Register auf **1** gesetzt wird, arbeitet der Prozessor in der BCD-Arithmetik; wenn dieses Bit auf **0** steht, wird binär gerechnet.

Befehl	Adressierarten				Condition Code Register					Befehlsablauf
	Unmittelbar	Seite Null	Index	Absolut	H	N	Z	V	C	
ADDA Oprd.	8B	9B	AB	BB	●	●	●	●	●	(A) + Oprd. → A
ADDB Oprd.	CB	DB	EB	FB	●	●	●	●	●	(B) + Oprd. → B
ADCA Oprd.	89	99	A9	B9	●	●	●	●	●	(A) + C + Oprd. → A
ADCB Oprd.	C9	D9	E9	F9	●	●	●	●	●	(B) + C + Oprd. → B
SUBA Oprd.	80	90	A0	B0	●	●	●	●	●	(A) − Oprd. → A
SUBB Oprd.	C0	D0	E0	F0	●	●	●	●	●	(B) − Oprd. → B
SUCA Oprd.	82	92	A2	B2	●	●	●	●	●	(A) − C − Oprd. → A
SUCB Oprd.	C2	D2	E2	F2	●	●	●	●	●	(B) − C − Oprd. → B
ANDA Oprd.	84	94	A4	B4		●	●	0		(A) ∧ Oprd. → A
ANDB Oprd.	C4	D4	E4	F4		●	●	0		(B) ∧ Oprd. → A
ORAA Oprd.	8A	9A	AA	BA		●	●	0		(A) ∨ Oprd. → A
ORAB Oprd.	CA	DA	EA	FA		●	●	0		(B) ∨ Oprd. → A
EORA Oprd.	88	98	A8	B8		●	●	0		(A) ⊻ Oprd. → A
EORB Oprd.	C8	D8	E8	E9		●	●	0		(B) ⊻ Oprd. → A
BYTE	2	2	2	3	=1	=1	=1	=1	=1	

Halb Carry, Negativ, Zero, Overflow, Carry

CONDITION CODE REGISTER: 1 1 H I N Z V C (7 6 5 4 3 2 1 0)

Bei Subtraktionen wird das Carry-Bit auf 1 gesetzt, wenn geborgt wurde.

Befehl	Code	Bytes	H	N	Z	V	C	Befehlsablauf
ABA	1B	1	●	●	●	●	●	(A) + (B) → A
SBA	10	1	●	●	●	●	●	(A) − (B) → A

Bild 4.7 Datenverarbeitungs-Befehle beim 6800

Die Adressierarten bei den Datenverarbeitungs-Befehlen entsprechen den Adressierarten bei den Datentransport-Befehlen, so daß sie hier

6502 nicht noch einmal erklärt zu werden brauchen.

6800 Da der *6800* zwei Akkumulatoren A und B besitzt und bei den Additions- und Subtraktionsbefehlen unterschieden wird, ob mit oder ohne Carry gearbeitet werden soll, gibt es bei diesem Prozessor insgesamt vierzehn verschiedene Datenverarbeitungs-Befehle mit zwei Operanden, wobei für jeden dieser Befehle vier Adressiermöglichkeiten bestehen (Bild 4.7).

Bei diesem Prozessor enthält das Programm-Status-Wort, hier CONDITION CODE genannt, mit einer Ausnahme nur Ergebnisse und keine Anweisungen. Ein Punkt in der Tabelle bedeutet wieder, daß das betreffende Bit durch das Ergebnis der Operation beeinflußt wird. Steht in der Tabelle eine **1**, so wird durch den Befehl immer eine **1** gesetzt, steht dort eine **0**, so wird diese Bit-Position durch den Befehl immer gelöscht.

Der im Befehl angezogene Akkumulator enthält immer den ersten Operanden und nimmt das Ergebnis der Operation auf. Der zweite Operand wird der Operandenadresse entnommen, die je nach Adressierart in gleicher Weise wie bei den Datentransport-Befehlen vom Prozessor gefunden wird.

Der *6800* hat auch je einen Befehl zum Addieren bzw. Subtrahieren

6800 beider Akkumulator-Inhalte.

8080 Ganz anders aufgebaut sind nun die Datenverarbeitungs-Befehle beim *8080*. Es gibt hier nur Ein-Byte-Befehle, und das Register A enthält immer einen Operanden und nimmt immer das Ergebnis auf. Der zweite Operand kann sich in einem der anderen Register befinden, wie dies im Bild 4.8 dargestellt ist.

Die schon erwähnte Besonderheit des Prozessors, mit indirekter Adressierung durch das Registerpaar HL zu arbeiten, erlaubt es auch hier, nach entsprechender Vorbereitung durch Programmbefehle als zweiten Operanden den Inhalt einer beliebigen Speicherstelle abzurufen. Hierfür sind die Befehlscode zu benutzen, die unter dem Buchstaben M in der Tabelle stehen.

Auch beim *8080* gibt es getrennte Befehle für Addition und Subtraktion mit und ohne Berücksichtigung des Carry-Bit, und das Programm-Status-Register enthält auch hier nur Ergebnisse und keine Anweisungen.

Außer den Datenverarbeitungs-Befehlen mit Register-Adressierung,

Ein - Byte - Befehle Adresse des 2. Operanden									FLAG - Register					Befehlsablauf
Befehl	A	B	C	D	E	H	L	M	S	Z	A	P	C	
ADD R	87	80	81	82	83	84	85	86	●	●	●	●	●	(A) + (R) → A
ADC R	8F	88	89	8A	8B	8C	8D	8E	●	●	●	●	●	(A) + C + (R) → A
SUB R	97	90	91	92	93	94	95	96	●	●	●	●	●	(A) - (R) → A
SBB R	9F	98	99	9A	9B	9C	9D	9E	●	●	●	●	●	(A) - C - (R) → A
ANA R	A7	A0	A1	A2	A3	A4	A5	A6	●	●		●	●	(A) ∧ (R) → A
ORA R	B7	B0	B1	B2	B3	B4	B5	B6	●	●		●	●	(A) ∨ (R) → A
XRA R	AF	A8	A9	AA	AB	AC	AD	AE	●	●		●	●	(A) ∀ (R) → A

S = 0 = Positiv
S = 1 = Negativ
Z = 0 = Null
Z = 1 = Nicht Null
A = 1 = Übertrag v. Bit3 in 4
P = 0 = Parität ungerade
P = 1 = Parität gerade
C = 1 = Übertrag

Vorzeichen - Sign.
Zero - Null
Hilfs - Carry
Parity
Carry

S	Z	0	A	0	P	1	C
7	6	5	4	3	2	1	0

FLAG - Zustandsregister

a)

Zwei - Byte - Befehle							
Befehl	Code	S	Z	A	P	C	Befehlsablauf
ADI Wert	C6	●	●	●	●	●	(A) + Wert → A
ACI Wert	CE	●	●	●	●	●	(A) + C + Wert → A
SUI Wert	D6	●	●	●	●	●	(A) - Wert → A
SBI Wert	DE	●	●	●	●	●	(A) - C - Wert → A

Bei Subtraktionen wird das Carry - Bit auf 1 gesetzt, wenn geborgt wurde.

b)

DAD Rp (HL) + (Rp) → HL

0	0	Rp	1	0	0	1

Registerpaar →	BC	CD	HL	SP
DAD Rp	09	19	29	39

c)

Bild 4.8 Datenverarbeitungs-Befehle beim 8080

die also nur eine Länge von einem Byte haben, gibt es bei diesem Prozessor auch noch vier Zwei-Byte-Befehle für die Datenverarbeitung mit unmittelbarer Adressierung. Bei diesen Befehlen wird das im zweiten Befehls-Byte stehende Datenwort unmittelbar zum Inhalt des Akkumulators hinzuaddiert oder von diesem abgezogen, und es gibt dabei wieder je einen Befehl, bei dem der Übertrag mit berücksichtigt wird, und einen,
8080 bei dem dieser unberücksichtigt bleibt.

4.4.2 Increment-, Decrement- und Löschbefehle

Mit Ausnahme des *2650,* bei dem es eine Reihe von Möglichkeiten gibt, Registerinhalte ohne einen besonderen Befehl zu erhöhen bzw. zu erniedrigen, haben alle hier besprochenen Prozessoren besondere Increment- und Decrementbefehle, durch die Inhalte von Registern oder Speicherstellen jeweils um den Wert **1** erhöht oder erniedrigt werden können. Es handelt sich hierbei um Datenverarbeitungs-Befehle, bei denen nur ein Operand beteiligt ist.

In diese Befehlsgruppe gehören auch die ähnlich aufgebauten Lösch- und Komplementier-Befehle, die es allerdings nicht bei allen Prozessoren gibt, da das Löschen einer Speicherstelle, d.h. das Nullsetzen aller Bit, das man auch mit CLEAR bezeichnet, auch durch einen unmittelbaren Ladebefehl mit **00** oder durch die logische UND-Verknüpfung mit der Maske **00** erreicht werden kann.

Das gleiche gilt bezüglich der Komplementier-Befehle, die nur der *6800* und der *8080* kennt. Auch hierfür wird bei den anderen Prozessoren die logische Verknüpfung «EXKLUSIV ODER» mit der Maske **11** verwendet.

6502 Der *6502* kennt Increment- und Decrementbefehle, die in Bild 4.9 dargestellt sind. Durch die Befehle **INC** und **DEC** wird immer eine Speicherstelle angesprochen, die auf vier verschiedene Arten adressiert werden kann. Für Index-Adressierung kann bei diesen Befehlen nur das Indexregister X verwendet werden.

Die beiden Indexregister können außerdem durch Ein-Byte-Befehle erhöht oder erniedrigt werden. Es gibt allerdings keine Befehle, mit
6502 denen der Akkumulator direkt incrementiert bzw. decrementiert werden kann.

6800 Der *6800* kennt eine größere Anzahl von Befehlen zur Veränderung eines Operanden, die in diese Gruppe fallen. Außer den Befehlen für die

Befehl	Seite Null	Seite Null, X	Absolut	Absolut, X	N	V	Z	C	Befehlsablauf
INC Adr.	E6	F6	EE	FE	•		•		(Adr.)+1 → Adr
DEC Adr.	C6	D6	CE	FE	•		•		(Adr.)-1 → Adr
BYTE	2	2	3	3					

Ein - Byte - Befehle						
Befehl	Code	N	V	Z	C	Ablauf
INX	E8	•		•		(X)+1 → X
INY	C8	•		•		(Y)+1 → Y
DEX	CA	•		•		(X)-1 → X
DEY	88	•		•		(Y)-1 → Y

Bild 4.9 Increment- und Decrement-Befehle beim 6502

Erhöhung und Erniedrigung des Inhalts gibt es Löschbefehle und Komplementier-Befehle. Die Befehle mit der Abkürzung **COM** bilden das Einer-Komplement, das heißt, aus jeder **0** wird eine **1** und aus jeder **1** eine **0**. Die mit **NEG** bezeichneten Befehle bilden das Zweier-Komplement, wie es im Abschnitt 2.2 beschrieben wurde.

Alle in diese Gruppe fallenden Befehle des *6800* sind in der Tabelle im Bild 4.10 dargestellt. Es ist zu erkennen, daß es eine Gruppe von Befehlen mit absoluter Adressierung bzw. Index-Adressierung gibt, durch die der Inhalt von Speicherstellen entsprechend verändert werden kann.

Bei der zweiten Befehlsgruppe handelt es sich um Befehle mit Register-Adressierung, durch die der Inhalt der beiden Akkumulatoren bzw. der Inhalt des Indexregisters und des Stapelzeigers verändert wird. ***6800***

Der *8080* kennt in dieser Befehlsgruppe nur einen Komplementier-Befehl, mit dem der Inhalt des Akkumulators komplementiert wird. Es können aber alle Register und auch alle Registerpaare durch Ein-Byte-Befehle mit Register-Adressierung incrementiert und decrementiert werden, wie es aus der Tabelle im Bild 4.11 ersichtlich ist. ***8080***

Die Befehle haben wieder im Operationscode ein Feld, in das die Codierung für das gewählte Register bzw. Registerpaar, wie schon mehrfach erklärt, einzutragen ist. Es gibt auch wieder die Möglichkeit der

Befehl	Register	Index	Absolut	H	N	Z	V	C	Befehlsablauf
INC Adr.		6C	7C		●	●	●		(Adr. +1) → Adr.
DEC Adr.		6A	7A		●	●	●		(Adr. −1) → Adr.
CLR Adr.		6F	7F		0	1	0	0	00 → Adr.
COM Adr.		63	73		●	●	0	1	($\overline{\text{Adr.}}$) → Adr.
NEG Adr.		60	70		●	●	●	●	($\overline{\text{Adr.}}$ +1) → Adr.
INCA	4C				●	●	●		(A + 1) → A
INCB	5C				●	●	●		(B + 1) → B
INX	08					●			(X + 1) → X
INS	31								(SP+1) → SP
DECA	4A				●	●	●		(A − 1) → A
DECB	5A				●	●	●		(B − 1) → B
DEX	09					●			(X − 1) → X
DES	34								(SP−1) → SP
CLRA	4F				0	1	0	0	00 → A
CLRB	5F				0	1	0	0	00 → B
COMA	43				●	●	0	1	($\overline{\text{A}}$) → A
COMB	53				●	●	0	1	($\overline{\text{B}}$) → B
NEGA	40				●	●	●	●	($\overline{\text{A}}$ + 1) → A
NEGB	50				●	●	●	●	($\overline{\text{B}}$ + 1) → B
BYTE	1	2	3						

Bild 4.10 Increment-, Decrement-, Lösch- und Complement-Befehle beim 6800

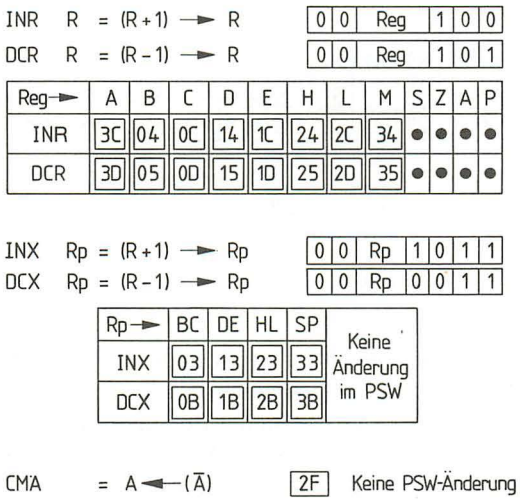


Bild 4.11 Increment-, Decrement- und Complementbefehle beim 8080

indirekten Adressierung, bei der die Speicherstelle angesprochen wird, deren Adresse im Registerpaar HL steht.

Beim *8080* wird der Inhalt des Programm-Status-Worts bei diesen Befehlen nicht verändert. **8080**

4.4.3 Rotations- und Schiebebefehle

Alle Prozessoren kennen in ihrem Befehlssatz Rotationsbefehle. Durch sie wird ein Bitmuster zyklisch verschoben, wie es schon im Abschnitt 3.6.3 vorgestellt wurde. Schiebebefehle gibt es allerdings nicht in allen Befehlssätzen.

Der *2650* kennt je einen Befehl für das Rotieren rechts und links herum. Wie aus Tabelle Bild 4.12 zu ersehen, gibt es insgesamt 8 verschiedene Befehlscodes, durch die der Inhalt der 4 Vielzweckregister jeweils um 1 Bit rechts oder links rotiert wird. **2650**

Der besondere Aufbau des unteren Programm-Status-Worts (PSL) erlaubt es nun, durch Setzen oder Löschen des Bit Nr. 3 festzulegen, ob das Carry-Bit in die Rotation einbezogen werden soll oder unberücksich-

Ein-Byte-Befehle									
	R0	R1	R2	R3	CC	IDC	OVF	C	
RRR,R	50	51	52	53	•	•	•	•	Reg. rechts rotieren
RRL,R	D0	D1	D2	D3	•	•	•	•	Reg. links rotieren

PSL

7	6	5	4	3	2	1	0	
						1		Rotation durch Carry
							0	Rotation ohne Carry

Bild 4.12 Rotationsbefehle beim 2650

tigt bleibt. Im ersten Fall ist im Bit Nr. 3 des PSL eine **1** zu setzen, im zweiten Fall muß dort eine **0** eingeschrieben werden.

Durch den Befehl werden die Bit des Bedingungscodes und die Ergebnis-Bit im unteren Programm-Status-Wort beeinflußt. Das Carry-Bit wird natürlich nur verändert, wenn die Rotation durch das Carry-Bit erfolgt ist.

2650
6502

Beim *6502* wird bei allen Rotationen das Carry-Bit mit eingeschlossen. Fünf Adressierarten, wie in Bild 4.13 erklärt, können angewendet werden. Außer dem Inhalt des Akkumulators können also auch die Inhalte von Speicherstellen rotiert werden.

Der Prozessor kennt auch zwei Schiebebefehle. Bei ihnen wird der Inhalt des Akkumulators oder der durch Adressierung angerufenen Speicherstelle jeweils um eine Position nach rechts oder links verschoben. Das vorderste Bit gelangt in das Carry, hinten wird eine **0** nachgeschoben.

	AKKU	Seite 0	Seite 0,X	Absolut	Absolut, X	N	Z	C	
ROR Oprd.	6A	66	76	6E	7E	•	•	•	Oprd. durch Carry rechts rotieren
ROL Oprd.	2A	26	36	2E	3E	•	•	•	Oprd. durch Carry links rotieren
ASL Oprd.	0A	06	16	0E	1E	•	•	•	C ← □□□□□□□□ ← 0
LSR Oprd.	4A	46	56	4E	5E	•	•	•	0 → □□□□□□□□ → C
BYTE	1	2	2	3	3	•	•	•	

Bild 4.13 Rotations- und Schiebebefehle beim 6502

Im Programm-Status-Register werden durch die Rotations- und Schiebebefehle die Negativ-, Null- und Carry-Bit verändert. **6502**

Auch beim *6800* wird das Carry-Bit immer in die Rotations- und Schiebevorgänge mit eingeschlossen. Insgesamt gibt es fünf Möglichkeiten für die Veränderung des Bitmusters. Vier davon entsprechen den Befehlen des *6502*. Beim Befehl **ASR** wird das Bitmuster auch nach rechts in das Carry hinein verschoben, jedoch wird hinten nicht mit einer Null aufgefüllt, sondern der vorher im Bit Nr. 7 stehende Inhalt wird wieder in das Bit Nr. 7 eingesetzt. **6800**

Durch Ein-Byte-Befehle kann jeweils der Inhalt der beiden Akkumulatoren rotiert oder verschoben werden. Mit Zwei-Byte-Befehlen kann über das Indexregister jede beliebige Adresse angesprochen und ihr Inhalt durch Rotieren oder Verschieben verändert werden. Das gleiche ist auch bei den Drei-Byte-Befehlen mit absoluter Adressierung möglich.

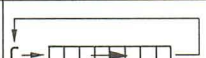
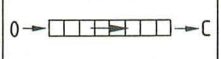
Befehl	Akku	Index	Absolut	N	Z	V	C	Befehlsausführung
ROR Oprd.		66	76	●	●	●	●	
RORA	46			●	●	●	●	
RORB	56			●	●	●	●	
ROL Oprd.		69	79	●	●	●	●	
ROLA	49			●	●	●	●	
ROLB	59			●	●	●	●	
ASL Oprd.		68	78	●	●	●	●	
ASLA	48			●	●	●	●	
ASLB	58			●	●	●	●	
ASR Oprd.		67	77	●	●	●	●	
ASRA	47			●	●	●	●	
ASRB	57			●	●	●	●	
LSR Oprd.		64	74	●	●	●	●	
LSRA	44			●	●	●	●	
LSRB	45			●	●	●	●	
Byte	1	2	3					

Bild 4.14 Rotations- und Schiebebefehle beim 6800

Im Programm-Status-Register werden alle Ergebnis-Bit, wie in der Tabelle im Bild 4.14 gezeigt, verändert. Wenn im Bit Nr. 7 nach dem Befehl eine **1** steht, erscheint auch im Negativ-Bit eine **1.** Eine **1** im Null-Bit zeigt, daß der Inhalt des zu verarbeitenden Byte **00** wurde. Die Veränderung des V-Bit ist in diesem Zusammenhang uninteressant. Die Veränderung des Carry-Bit ergibt sich aus dem Bitmuster, das durch den
6800 Befehl verarbeitet wird.

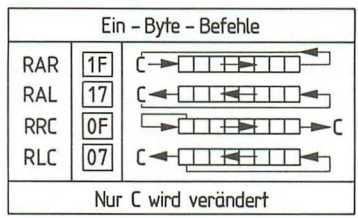


Bild 4.15 Rotationsbefehle beim 8080

8080 Der *8080* kennt nur Rotationsbefehle, durch die der Inhalt des Akkumulators verändert werden kann (Bild 4.15). Es stehen vier Ein-Byte-Befehle zur Verfügung mit Berücksichtigung des Carry. Der Befehl **RRC** entspricht dem Schiebebefehl **ASR** des *6800*. Bei dem Befehl **RRL** findet eine Rotation ohne Berücksichtigung des Carry statt. Zusätzlich wird der Inhalt des Bit Nr. 7 in das Carry hineingeschoben.

Bei allen Befehlen wird im Programm-Status-Wort nur das Carry-Bit
8080 verändert.

4.5 Ein- und Ausgabebefehle

Von den hier beschriebenen Prozessoren kennen nur der *2650* und *8080* besondere Ein- und Ausgabebefehle. Bei den anderen Prozessoren werden die Ein-/Ausgabebausteine wie Speicherplätze behandelt und mit den üblichen Befehlen, durch die Speicherplätze adressiert werden, angesprochen.

Im Grundsatz ist diese Arbeitsweise, die man auf englisch auch MEMORY MAPPED I/O nennt, auch beim *2650* und beim *8080* mög-

lich. Darüber hinaus erlauben diese beiden Prozessoren jedoch noch die Bedienung von bis zu 256 Ein- und Ausgabeport außerhalb des Speicherraums durch besondere Zwei-Byte-Befehle.

Der *8080* braucht für Ein- und Ausgaben nur die Befehle **IN** für Ein- **8080**
gaben und **OUT** für Ausgaben. Im zweiten Befehls-Byte wird dann die Port-Nummer angeführt, die zwischen **00** und **FF** lauten kann (Bild 4.16).

IN Port - Adr.	A ← (Port)	DB	Port - Adr.
OUT Port - Adr.	Port ← (A)	D3	Port - Adr.
Port - Adr. nur 00...FF			

Bild 4.16 Befehle für Ein- und Ausgaben beim 8080

Ein- bzw. Ausgaben erfolgen immer in den bzw. aus dem Akkumulator. **8080**

Ein- und Ausgabebefehle mit gleicher Wirkung und gleichem Aufbau **2650**
gibt es auch beim *2650*. Ausgeben wird hier mit *Schreiben* (WRITE) und Eingeben mit *Lesen* (READ) bezeichnet. Grundsätzlich können bei diesem Prozessor die Inhalte der vier Register R0 bis R3 ausgegeben werden; ebenso auch wie diese Register 8 Bit breite Eingaben aus einem Eingangsport übernehmen können. Die Befehlscodes sind aus der Tabelle in Bild 4.17 zu ersehen.

Byte	Befehl	R0	R1	R2	R3	Befehlsablauf
1	WRTD,R	F0	F1	F2	F3	(R) → Port D
1	REDD, R	70	71	72	73	R ← (Port D)
1	WRTC,R	B0	B1	B2	B3	(R) → Port C
1	REDC,R	30	31	32	33	R ← (Port C)
2	WRTE,R Adr.	D4	D5	D6	D7	(R) → Port Adr.
2	REDE,R Adr.	54	55	56	57	R ← (Port Adr.)

Port Adr. nur 00 ... FF; Bei Eingaben wird CC beeinflußt

Bild 4.17 Befehle für Ein- und Ausgaben beim 2650

Außer den erwähnten Zwei-Byte-Befehlen **WRTE** und **REDE,** bei denen das **E** für EXTENDED steht und mit denen im zweiten Befehlsbyte 256 verschiedene Port angesprochen werden können, gibt es noch zwei besondere Ein- und Ausgabebefehle mit nur einem Byte Länge.

Sie bedienen zwei hardwaremäßig besonders leicht anzuschließende Port, die, ohne daß dies eine besondere Bedeutung hat, **Data** und **Control** genannt werden. Ausgaben in diese Port und Eingaben aus diesen Port sind auch aus bzw. in alle Register möglich.

4.6 Vergleichs- und Testbefehle

Wie bereits in Abschnitt 3.4.3 gezeigt, ähneln die Vergleichs- und Testbefehle den Datenverarbeitungs-Befehlen mit zwei Operanden. Bei Vergleichsbefehlen wird eine Subtraktion durchgeführt, während bei Testbefehlen eine logische UND-Verknüpfung erfolgt.

Wichtig ist es, daß der Inhalt beider Operanden bei diesen Befehlen bestehen bleibt. Das eigentliche Ergebnis der Operation wird nicht abgespeichert. Dafür werden je nach dem Ausfall des Ergebnisses gewisse Bedingungs-Bit gesetzt oder gelöscht, so daß die beiden Operanden auch noch für weitere Programmbefehle zur Verfügung stehen.

Vergleichsbefehle sind bei allen hier besprochenen Mikroprozessoren ähnlich aufgebaut, sie unterscheiden sich nur bezüglich der möglichen Operanden.

6502 Beim *6502* wird jeweils der Inhalt des Akkumulators, des Indexregisters X und des Indexregisters Y mit einem anderen Bitmuster, das auf verschiedene Weise als Operand adressiert werden kann, verglichen, wie es in der Tabelle in Bild 4.18 dargestellt ist.

Wichtig für alle Vergleichsoperationen ist vor allen Dingen die unmittelbare Adressierung mit Zwei-Byte-Befehlen, bei der im zweiten Byte vom Programmierer eine Maske programmiert werden kann, mit der der Vergleich dann durchgeführt wird.

Nach dem Befehl wird im Programm-Status-Register das Negativ-Bit auf **1** gesetzt, wenn durch die Operation ein negativer Zahlenwert im Zweier-Komplementsystem entstanden ist. Das Null-Bit (ZERO) wird auf **1** gesetzt, wenn das Ergebnis des Vergleichs Null war, d.h., wenn beide

6502 Operanden den gleichen Wert hatten.

6800 Die Vergleichsbefehle des *6800* sind in gleicher Weise aufgebaut. Auch hier werden einzelne Bit des Programm-Status-Worts entsprechend dem Ergebnis der Operation verändert.

Zusätzlich gibt es bei diesem Prozeß noch einen Ein-Byte-Befehl mit dem Code **CBA,** durch den der Inhalt der beiden Akkumulatoren mit-

6800 einander verglichen wird.

Befehl	Unmittelbar	Absolut	Seite Null	(Indirekt,X)	(Indirekt),Y	Seite Null,X	Absolut, X	Absolut,Y	N	Z	C	Befehls-ablauf
CMP Oprd.	C9	CD	C5	C1	D1	D5	DD	D9	●	●	●	(A) - Oprd.
CPX Oprd.	E0	EC	E4						●	●	●	(X) - Oprd.
CPY Oprd.	C0	CC	C4						●	●	●	(Y) - Oprd.
BYTE	2	3	2	2	2	2	3	3				

6502

Befehl	Unmittelbar	Seite Null	Index	Absolut	N	Z	V	C	Befehls-ablauf
CMPA Oprd.	81	91	A1	B1	●	●	●	●	(A) - Oprd.
CMPB Oprd.	C1	D1	E1	F1	●	●	●	●	(B) - Oprd.
CPX Oprd.	8C	9C	AC	BC	●	●	●	●	(X) - Oprd., Oprd.+1
BYTE	2	2	2	3					
CBA			11		●	●	●	●	(A) - (B)

6800

Befehl	A	B	C	D	E	H	L	M	S	Z	A	P	C	Ablauf
CMP R	BF	B8	B9	BA	BB	BC	BD	BE	●	●	●	●	●	(A) - (R)

CPI Wert (A) - Wert FE Wert PSW wie CMP

8080

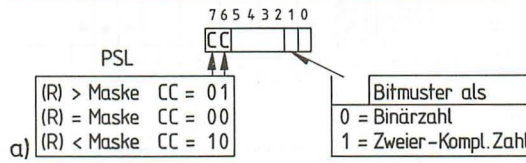
Bild 4.18 Vergleichsbefehle bei den Prozessoren 6502, 6800 und 8080

Der *8080* erlaubt mit Ein-Byte-Befehlen jeweils den Vergleich des Inhalts des Akkumulators mit einem der Register. Wie bei diesem Prozessor üblich, kann auch der Vergleich des Inhalts, dessen Adresse im Registerpaar HL steht, mit dem Akkumulator erfolgen. **8080**

Es gibt auch noch einen Zwei-Byte-Befehl, mit dem der Akkumulatorinhalt unmittelbar mit der Maske im zweiten Befehls-Byte verglichen wird. **8080**

B	Befehl	R0	R1	R2	R3	Befehlsablauf
1	COMZ, R	E0	E1	E2	E3	(R) ≈ (R0)
2	COMI, R Wert	E4	E5	E6	E7	(R) ≈ Wert
2	COMR, R Offset	E8	E9	EA	EB	(R) ≈ (PC + 0t)
3	COMA, R Adr.	EC	ED	EE	EF	(R) ≈ (Adr)

a)



B	Befehl	R0	R1	R2	R3	Befehlsablauf
2	TMI, R Maske	F4	F5	F6	F7	(R) ∧ Maske

CC = 00 wenn alle '1' in Maske und Reg. übereinstimmen
CC = 01 wenn nicht alle '1' übereinstimmen

b)

z.B.	Register:	01001100	01001100
	Maske	11001100	01001100
	CC	01	00

Bild 4.19 Vergleichs- und Testbefehle beim 2650

2650 Etwas komplizierter im Aufbau sind die Vergleichsbefehle beim *2650*. Es gibt vier verschiedene Befehle, die in der Tabelle in Bild 4.19 dargestellt sind.

Beim Befehl **COMZ** wird der Inhalt eines der Vielzweckregister mit dem Inhalt des Registers 0 verglichen. Es handelt sich um einen Ein-Byte-Befehl. Der Zwei-Byte-Befehl **COMI** vergleicht den Inhalt eines beliebigen Registers mit dem Bitmuster, das als Maske im zweiten Befehls-Byte steht.

Bei den Befehlen **COMR** und **COMA** werden die Registerinhalte mit dem Inhalt der Speicherstellen, die durch relative oder absolute Adressierung zu finden sind, verglichen. Wie bei diesem Prozessor üblich, kann bei der relativen Adressierung auch noch ein Indexregister bestimmt werden. Dann steht der Operand des Vergleichs im Register 0. Beim Vergleichsbefehl mit absoluter Adressierung ist sowohl Index- als auch indirekte Adressierung zusätzlich möglich.

Abweichend von den anderen Prozessoren ist die Meldung der Ergebnisse der Vergleichsoperationen. Grundsätzlich werden hierfür die Bit Nr. 6 und 7 des unteren Programm-Status-Wortes verwendet, die in der Sprache des Prozessors CONDITION CODE (auf deutsch *Bedingungscode*) genannt werden. Für die beiden Bedingungscode-Bit gilt die in der Tabelle in Bild 4.19 gezeigte Verschlüsselung. Wenn der Inhalt des zu prüfenden Registers größer ist als das Bitmuster im Vergleichsregister, der Wert der Maske oder der Inhalt der angesprochenen Adresse, so wird als Bedingungscode **01** gesetzt. Sind beide Bitmuster gleich, erscheint dort **00,** und wenn das zu prüfende Register kleiner ist, wird dort **10** eingesetzt.

Eine weitere Besonderheit besteht darin, daß durch Setzen oder Löschen des Bit Nr. 1 im unteren Programm-Status-Wort noch festgelegt werden kann, ob der Prozessor beim Vergleich die Zahlen als reine Binärzahlen anzusehen hat oder aber sie als Zahlen im Zweier-Komplementsystem behandeln soll. Im ersten Fall würde z.B. die Binärzahl **1000 0000** größer sein als die Zahl **0100 0000.** Dagegen wäre als Zweier-Komplementzahl betrachtet die erstgenannte Zahl, die einen negativen Wert darstellt, kleiner als die zweitgenannte.

Der *2650* hat noch einen besonderen Testbefehl, mit dem es möglich ist, festzustellen, ob einzelne Bit in einem Bitmuster auf **1** gesetzt sind oder nicht. Auch dieser Befehl erlaubt es, die Inhalte aller Register zu überprüfen.

Es handelt sich um einen Zwei-Byte-Befehl, bei dem in den zweiten Befehls-Byte eine Maske aufgenommen werden muß. Der Befehl führt eine logische UND-Verknüpfung der Maske mit dem Registerinhalt aus. Danach wird der Bedingungscode auf **00** gesetzt, wenn sich in allen Bit-Positionen, in denen in der Maske eine **1** steht, auch im zu prüfenden Bitmuster eine **1** befindet. Ist dies nicht der Fall, wird **01** in den Bedingungscode eingeschrieben.

2650

6502

Auch beim *6502* gibt es Befehle, die die Prüfung einzelner Bit durch logische UND-Verknüpfung mit Hilfe einer Maske ermöglichen. Bei diesem Prozessor wird allerdings, wie aus Bild 4.20 zu ersehen, kein Befehl mit unmittelbarer Adressierung hierfür verwendet, sondern die Maske muß im Speicherraum stehen und kann dann entweder durch absolute Adressierung oder durch Adressierung auf Seite 0 angesprochen werden. Überprüft wird grundsätzlich der Inhalt des Akkumulators.

Das Ergebnis der Prüfung erscheint im Null-Bit des Programm-Status-

BIT Adr.	(A) ∧ (Adr.)	2C	Unt. Adr.	Ob. Adr.
BIT Adr. S. Null	(A) ∧ (Adr.)	24	Adr.	

Z=1 wenn 00. Bit 6 (Adr.) ⟶ V. Bit 7 von (Adr.) ⟶ N

```
z.B. (Akku)     10001100    00001100
     ( Adr. ) ∧ 01001100    11000000
                00001100    00000000
          V  =  └──► = 1     └──► 1
          N  =  └──► = 0     └──► 1
          Z  =       = 0          1
```

Bild 4.20 Testbefehle beim 6502

Worts, wo immer dann eine **1** gesetzt wird, wenn die **1** in den beiden Bitmustern übereinstimmt.

Zusätzlich wird nun bei diesem Prozessor der Inhalt des Bit Nr. 6 aus der angesprochenen Adresse in das Überlauf-Bit und der Inhalt von Bit Nr. 7 in das Negativ-Bit des Programm-Status-Worts eingeschrieben.

6502 Die gleichen Befehle können auch benutzt werden, wenn man den Inhalt einer Adresse überprüfen will. In diesem Fall muß vorher die Prüfmaske in den Akkumulator geladen werden.

6800 Auch der *6800* kennt Testbefehle. Bei den Befehlen **BITA** und **BITB** wird auch zwischen dem Inhalt eines Akkumulators und dem Inhalt des

	Akkumulator	Unmittelbar	Seite Null	Index	Absolut	N	Z	V	C	
BITA Oprd.		85	95	A5	B5	•	•	○		(A) ∧ Oprd.
BITB Oprd.		C5	D5	E5	F5	•	•	○		(B) ∧ Oprd.
TST Oprd.				6D	7D	•	•	○	○	Oprd. - 00
TSTA	4D					•	•	○	○	(A) - 00
TSTB	5D					•	•	○	○	(B) - 00
BYTE	1	2	2	2	3					

Bild 4.21 Testbefehle beim 6800

durch Adressierung gewonnenen Operanden eine UND-Verknüpfung ausgeführt.

Ist das Ergebnis **00**, so wird das Null-Bit im Programm-Status-Wort auf **1** gesetzt.

Darüber hinaus gibt es noch weitere Testbefehle, bei denen nur vom Operanden bzw. vom Akkumulator der Wert **00** subtrahiert wird. Dies hat den Zweck, daß nach dem Befehl das Negativ-Bit und das Null-Bit im Programm-Status-Wort entsprechend dem Inhalt des geprüften Bitmusters gesetzt oder gelöscht werden (Bild 4.21). **6800**

4.7 Status-Register-Befehle

Die Programm-Status-Worte der Mikroprozessoren *6502, 6800* und *8080* sind sehr ähnlich aufgebaut, so daß die Befehle, mit denen die einzelnen Bit in den Programm-Status-Worten verändert werden können, zusammen besprochen werden sollen.

Das Bild 4.22 zeigt den Aufbau der Programm-Status-Worte bei den besprochenen Prozessoren und daneben jeweils die Ein-Byte-Befehle, mit denen einzelne Bit gesetzt bzw. gelöscht werden können.

Im Bild sind die einzelnen Bit-Positionen in den Programm-Status-Worten mit den Originalbezeichnungen der Hersteller gekennzeichnet, und es ist vermerkt, bei welcher Bedingung oder Arbeitsweise dort eine **1** eingeschrieben wird.

Leider sind sich die Hersteller bezüglich der zu wählenden Worte nicht ganz einig gewesen, aber man erkennt leicht, daß z.B. das Negativ-Bit beim *8080* SIGN-Bit, also Vorzeichen-Bit, genannt wird.

Während im Programm-Status-Wort des *8080* nur Ergebnisse gemeldet werden, erhalten die Programm-Status-Worte der beiden anderen Prozessoren auch Anweisungen für die Arbeit des Prozessors. So kann durch Setzen einer **1** beim *6502* in der Bit-Position Nr. 2 und beim *6800* in der Bit-Position Nr. 4 ein Interrupt, also eine von außen eingeleitete Programmunterbrechung, verhindert werden.

Als Besonderheit erlaubt der *6502* durch Setzen einer 1 in der Bit-Position Nr. 4 eine Verarbeitung von Zahlen im BCD-System, ohne daß dafür wie bei den anderen Prozessoren noch besondere Korrekturbefehle notwendig sind. Hierauf wird im einzelnen noch im Abschnitt 4.10 eingegangen.

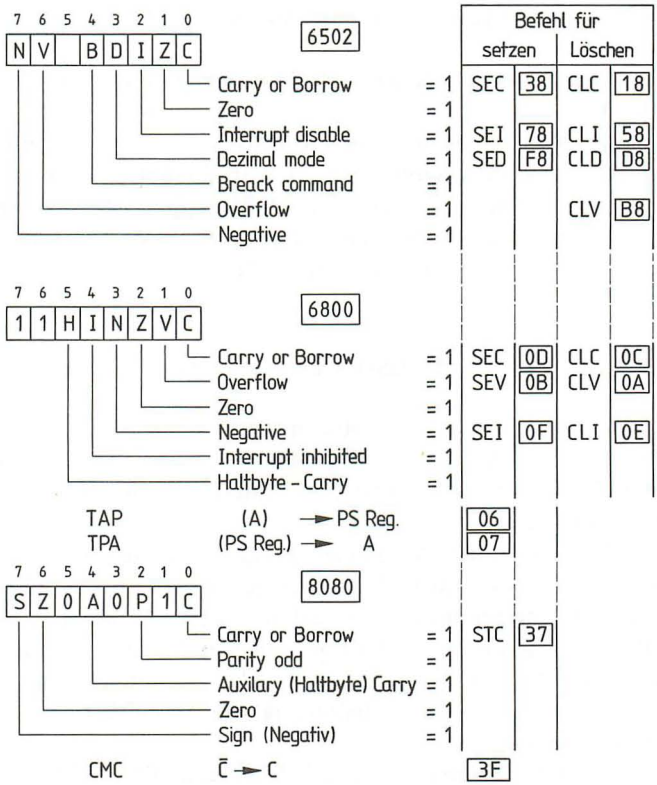


Bild 4.22 Aufbau der Programm-Status-Register und der Befehle für das Setzen und Löschen einzelner Bit in diesen Registern bei den Prozessortypen 6502, 6800 und 8080

Wie aus Bild 4.22 ersichtlich, können bei den genannten Prozessoren nicht alle Bit des Programm-Status-Worts durch Befehle verändert werden. Der *8080* kennt sogar nur einen Befehl in dieser Gruppe. Wenn das Carry-Bit gelöscht, also dort der Wert **0** eingeschrieben werden soll, muß zunächst der Befehl **STC** gegeben werden, der das Carry-Bit auf **1** setzt und danach durch den Befehl **CMC** der Inhalt des Carry-Bits komplementiert werden.

Völlig anders aufgebaut sind die Befehle, mit denen die Programm-Status-Worte beim *2650* zu bedienen sind. Wie Bild 4.23 zeigt, gibt es bei

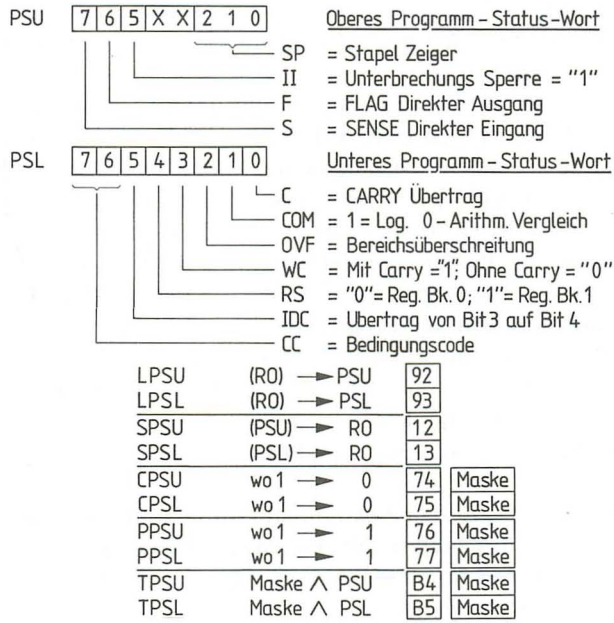


Bild 4.23 Aufbau der beiden Register für die Programm-Status-Worte beim 2650 und der Befehle für die Veränderung dieser Register

diesem Prozessor zwei Programm-Status-Worte, die PSU = PROGRAM STATUS UPPER, gleich oberes Programm-Status-Wort, und PSL = PROGRAM STATUS LOWER, gleich unteres Programm-Status-Wort, genannt werden.

Die Bit Nr. 0 bis 2 im oberen Programm-Status-Wort dienen als Stapelzeiger in Verbindung mit der Speicherung der Rückkehradressen bei der Unterprogramm-Verarbeitung, worauf im Abschnitt 4.9.3 noch näher eingegangen werden wird.

Das Bit Nr. 5 dient dazu, eine von außen kommende Unterbrechung zu verhindern, wenn es auf **1** gesetzt ist.

Die Bit Nr. 6 und 7 sind unmittelbar mit Hardwarekontakten des Prozessors verbunden, wodurch auf einfache Weise serielle Ein- und Ausgabeoperationen ermöglicht werden.

Im unteren Programm-Status-Wort werden die gleichen Ergebnisse

gemeldet wie bei den anderen Prozessoren. Darüber hinaus werden in diesem Wort aber auch noch Anweisungen an dem Prozessor festgelegt.

Noch nicht erklärt wurde die Bedeutung des Bit Nr. 4 mit der Bezeichnung RS (REGISTERBANK SELECT), gleich *Registerbank-Anwahl.* Wie aus Bild 4.2 ersichtlich, sind beim *2650* die Register 1 bis 3 doppelt vorhanden. Eine Gruppe gehört zur Registerbank **0** und die andere zur Registerbank **1.** Je nachdem, ob in das Bit Nr. 4 des unteren Programm-Status-Worts eine **0** oder eine **1** eingeschrieben ist, steht dem Programmierer entweder die Registerbank **0** oder die Registerbank **1** zur Verfügung. Durch Ändern des Programm-Status-Worts kann man also bei diesem Prozessor sehr schnell von einer Registerbank auf die andere übergehen und so z. B. den Inhalt der Register retten und dann die Register der anderen Bank weiterverwenden.

Für die Änderung des Inhalts der Programm-Status-Worte gibt es zwei Möglichkeiten.

Man kann zunächst den Akkumulator durch unmittelbares Adressieren mit dem gewünschten Inhalt laden und dann einen der **LPS**-Befehle verwenden, um den Inhalt des Akkumulators in das Programm-Status-Register zu transportieren.

Die zweite Möglichkeit besteht in der Verwendung der **CPS-** und der **PPS**-Befehle. Hierbei handelt es sich um Zwei-Byte-Befehle mit einer Maske im zweiten Befehls-Byte. Bei den **CPS**-Befehlen wird überall dort im Programm-Status-Wort eine **0** gesetzt, wo sich in der Maske eine **1** befindet. Umgekehrt wird durch den **PPS**-Befehl überall dort eine **1** eingesetzt, wo in der Maske auch eine **1** steht. Weiterhin gibt es noch die Prüfbefehle **TPS,** die in gleicher Weise mit einer UND-Verknüpfung arbeiten, wie es im Abschnitt 4.6 beschrieben wurde.

Dann gibt es noch die beiden **SPS**-Befehle, mit denen der Inhalt der Programm-Status-Worte in den Akkumulator transportiert werden kann.

4.8 Befehle zur Bedienung des Stapelspeichers

Während sich der Stapelspeicher beim *2650* direkt auf dem Prozessor-Chip befindet und nur für die Speicherung der Rücksprung-Adressen von Unterprogrammen verwendet wird, haben die anderen drei hier besprochenen Mikroprozessoren besondere Befehle in ihrem Befehlssatz, mit denen ein in einem RAM angeordneter Stapelspeicher bedient werden kann.

Ein Stapelspeicher (englisch STACK) wird immer von dem höchstwertigsten Speicherplatz beginnend in abnehmender Richtung gefüllt, wobei der zuletzt abgespeicherte Wert als erstes wieder zur Ausgabe gelangt.

Die Befehle zum Abspeichern in den Stapelspeicher werden meistens mit **PUSH**, die Befehle zum Auslesen von Informationen aus diesem Speicher entweder mit **PULL** oder **POP** bezeichnet.

Die genannten Prozessoren haben jeweils ein Register, das als Stapelzeiger (englisch STACK POINTER – SP) bezeichnet wird. Die in diesem Register stehende Information stellt die Adresse dar, in die als nächstes durch einen **PUSH**-Befehl abgespeichert oder aus der durch einen **PULL**-Befehl ein Bitmuster entnommen werden kann.

Da der *6502* nur ein 8 Bit breites Stapelzeiger-Register besitzt, muß der Stapelspeicher bei diesem Prozessor im Adreßbereich zwischen **0100** und **01FF** angeordnet werden. Die beiden anderen Prozessoren haben 16 Bit

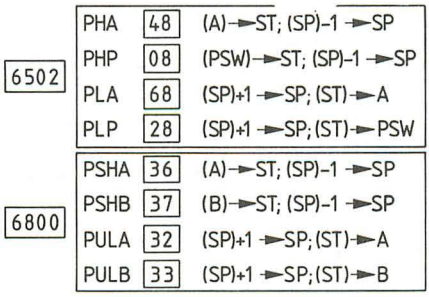


Bild 4.24 Befehle für Bedienung des Stapelspeichers beim 6502 und 6800

breite Stapelzeiger, so daß bei ihnen der Stapelspeicher an beliebiger Stelle des Adreßbereichs untergebracht werden kann.

6502 Beim *6502* läßt sich der Inhalt des Akkumulators und der Inhalt des Programm-Status-Wortes im Stapelspeicher abspeichern und von dort wieder zurückladen. Die einzelnen Befehle sind im Bild 4.24 dargestellt. Es handelt sich um Ein-Byte-Befehle, bei denen mit dem Speicher- oder Ladevorgang zusammen auch immer der Inhalt des Stapelzeigers um den Wert **1** vermindert oder erhöht wird. Beim Abspeichern erfolgt die Verminderung nach dem Speichervorgang. Beim Entnehmen aus dem Speicher wird der Stapelzeigerinhalt vor dem Entnahmevorgang um den

6502 Wert **1** erhöht.

6800 In praktisch gleicher Weise arbeitet der *6800*. Nur kann hier der gesamte Adreßraum für den Stapelspeicher verwendet werden, da der Stapelzeiger 16 Bit aufnimmt. Bei diesem Prozessor können die Inhalte der beiden Akkumulatoren im Stapelspeicher abgelegt und von dort

6800 wieder entnommen werden.

8080 Etwas anders arbeitet der Stapelspeicher beim *8080*, da hier mit einem

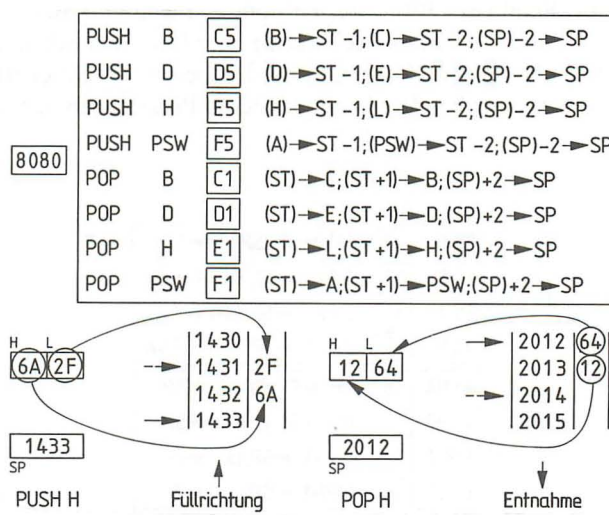


Bild 4.25 Befehle für die Bedienung des Stapelspeichers mit einem Beispiel für den 8080

Befehl jeweils der Inhalt eines Registerpaares abgespeichert wird. Hierfür werden immer zwei Speicherplätze benötigt. Wie es im Bild 4.25 dargestellt ist, wird beim Abspeichern, also beim **PUSH**-Vorgang, der Inhalt des höherwertigen Registers in die Speicherstelle geladen, die dem Wert des Stapelzeigers **−1** entspricht. Der Inhalt im niederwertigen Teil des Registerpaars kommt dann in die nächstniedrigere Speicherstelle. Nach dem Abspeichervorgang wird der Inhalt des Stapelzeigers um **2** vermindert.

Beim Auslesen aus dem Stapelspeicher, also dem **POP**-Vorgang, wird zunächst der Inhalt der Speicherstelle, auf die der Stapelzeiger zeigt, in den niederwertigen Teil des Registerpaars geladen und danach der Inhalt der nächsthöheren Speicherstelle in den höherwertigen Teil des Registerpaars transportiert. Anschließend erfolgt die Erhöhung des Stapelzeigers um den Wert **2.** ***8080***

4.9 Programmablaufbefehle

Mit der dritten großen Befehlsgruppe wird der normalerweise in einer Folge (man nennt das auch sequentiell) ablaufende Befehlsgang der Mikroprozessoren durch im Programm festgelegte Sprünge verändert. Befehle, die den Programmablauf beeinflussen, sind ein besonders wichtiges Hilfsmittel für effektive Programmgestaltung, wenn sie richtig angewandt werden. In erster Linie sollen darum die bedingten Sprünge verwendet werden, die verschiedene Programmabläufe nach einer vom Mikroprozessor getroffenen Entscheidung erlauben. Die sogenannten unbedingten Sprünge sollen bei der Programmierung möglichst wenig verwendet werden, denn jeder nicht durch die Programmstruktur notwendige Sprungbefehl macht das Programm nur unübersichtlicher.

4.9.1 Absolute Sprungbefehle

Da die unbedingten Sprungbefehle der verschiedenen Prozessoren recht ähnlich sind, sollen sie hier gemeinsam besprochen werden. Je nach Hersteller werden im Assemblerausdruck für Sprungbefehle die Abkürzungen der englischen Worte JUMP oder BRANCH verwendet.

Wie die Tabelle in Bild 4.26 zeigt, kennen alle Prozessoren unbedingte Sprungbefehle mit absoluter Adressierung. Dies bedeutet, daß zur Aus-

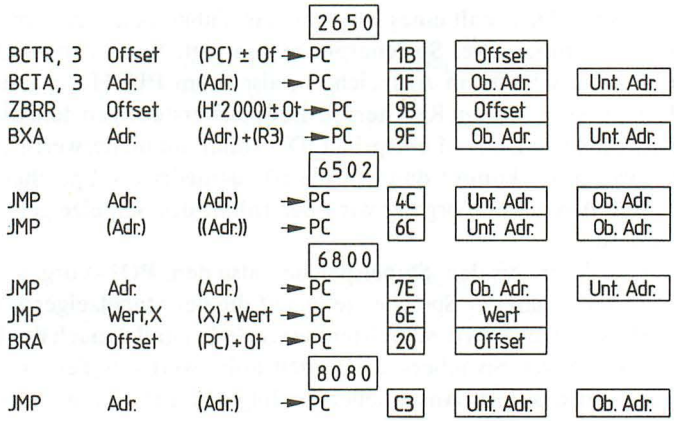


Bild 4.26 Unbedingte Sprungbefehle bei den Prozessortypen 2650, 6502, 6800 und 8080

führung des Sprunges der Inhalt des Befehlsadreß-Registers (PC) durch den Inhalt der im zweiten und dritten Befehls-Byte angesprochenen Speicherstelle ersetzt wird. Da in diesem Register immer die Adresse des nächsten auszuführenden Befehls steht, wird bei einem unbedingten Sprungbefehl immer der Programmbefehl als nächster ausgeführt, der in der Sprungadresse enthalten ist.

Der *2650* und der *6800* kennen auch Zwei-Byte-Befehle mit relativer Adressierung, bei denen also die Sprungadresse durch Addition des Befehlsadreß-Zählerinhalts und der im Befehl enthaltenen relativen Verschiebung (OFFSET) ermittelt wird.

Index-Adressierung ist beim *2650* mit dem Sprungbefehl **BXA** möglich. Auch der *6800* kennt einen Sprungbefehl mit indexierter Adressierung.

Beim *2650* gibt es noch den Befehl **ZBRR** mit relativer Adressierung, bei dem die relative Adresse immer ausgehend von der Adresse **2000** berechnet wird.

Die Bedingungs-Bit im Programm-Status-Wort werden selbstverständlich bei den Sprungbefehlen nicht geändert.

Sprung, wenn	6502 Relativ 2 Byte		6800 Relativ 2 Byte		8080 Absolut 3Byte	
Carry =0	BCC	90	BCC	24	JNC	D2
Carry =1	BCS	B0	BCS	25	JC	DA
Zero =0	BNE	D0	BNE	26	JNZ	C2
Zero =1	BEQ	F0	BEQ	27	JZ	CA
Positiv	BPL	10	BPL	2A	JP	F2
Negativ	BMI	30	BMI	2B	JM	FA
Overflow =0	BVC	50	BVC	28		
Overflow =1	BVS	70	BVS	29		
Parity =0					JPO	E2
Parity =1					JPE	EA
Größer als 0			BGE	2C		
Größer = 0			BGT	2E		
Größer			BHI	22		
Kleiner als 0			BLE	2F		
Kleiner			BLS	23		
Kleiner = 0			BLT	2D		

Bild 4.27 Bedingte Sprungbefehle bei den Prozessortypen 6502, 6800 und 8080

4.9.2 Bedingte Sprungbefehle

Bei den Mikroprozessoren *6502, 6800* und *8080* gibt es jeweils eine ganze Anzahl Befehle, durch die bedingte Sprünge veranlaßt werden können. Alle bei diesen drei Prozessoren im Befehlssatz enthaltenen Befehle sind in der Tabelle in Bild 4.27 dargestellt.

Grundsätzlich werden bei diesen Prozessoren die einzelnen Bit der Programm-Status-Worte, die als Ergebnis einer Operation gesetzt oder gelöscht werden, als Kriterien, ob ein Sprung durchzuführen ist, verwendet. Wenn die Bedingung erfüllt ist, erfolgt ein Programmsprung, wenn die Bedingung nicht erfüllt ist, wird der nächstfolgende Befehl im Programm ausgeführt.

Völlig anders, aber dafür auch sehr viel vielseitiger ist der Aufbau der Sprungbefehle beim 2650. Es gibt hier drei Gruppen von Befehlen, die auf unterschiedliche Bedingungen reagieren (Bild 4.28). **2650**

Bei einer Reihe von Befehlen wird der Programmsprung ausgeführt, wenn das im Befehl angezogene Register nicht den Wert Null hat. Jedes

	Sprung, wenn Register nicht Null ist					
B	Register→	R0	R1	R2	R3	Befehlsablauf
2	BRNR, R Offset	58	59	5A	5B	(PC) + Of → PC
3	BRNA, R Adr.	5C	5D	5E	5F	(Adr.) → PC
2	BIRR, R Offset	D8	D9	DA	DB	(PC) + Of + 1 → PC
3	BIRA, R Adr.	DC	DD	DE	DF	(Adr.) + 1 → PC
2	BDRR, R Offset	F8	F9	FA	FB	(PC) + Of - 1 → PC
3	BDRA, R Adr.	FC	FD	FE	FF	(Adr.) - 1 → PC

Sprung, wenn Bedingungscode übereinstimmen						
Bedingung-Kennz. im Assembler Befehl→ Inhalt der CC Bit →			0 00	1 01	2 10	
2	BCTR, B	Offset	18	19	1A	(PC) + Of → PC
3	BCTA, B	Adr.	1C	1D	1E	(Adr.) → PC

Sprung, wenn Bedingungscode nicht übereinstimmen						
Bedingung-Kennz. im Assembler Befehl→ Inhalt der CC Bit →			0 00	1 01	2 10	
2	BCFR, B	Offset	98	99	9A	(PC) + Of → PC
3	BCFA, B	Adr.	9C	9D	9E	(Adr.) → PC

Bild 4.28 Bedingte Sprungbefehle beim 2650

der vier Register kann als Bezugsregister für den Befehl verwendet werden.

In dieser Gruppe gibt es insgesamt sechs Befehle, je 3 davon mit relativer und 3 mit absoluter Adressierung. Während bei den Befehlen **BRN** lediglich der Sprung programmiert wird, erfolgt bei den Befehlen **BIR** noch zusätzlich eine Incrementierung des angezogenen Registers und bei den Befehlen **BDR** zusätzlich eine Decrementierung. Die Erhöhung bzw. Erniedrigung erfolgt, bevor die Sprungadresse berechnet wird. Mit Hilfe dieser Befehle ist es besonders einfach, Schleifenoperationen zu programmieren, wie später noch gezeigt werden wird.

Bei der zweiten Gruppe von bedingten Sprungbefehlen wird der Sprung ausgeführt, wenn der Inhalt des Bedingungscodes im unteren

Programm-Status-Wort, der ggf. vor dem Sprungbefehl durch einen Vergleichsbefehl (Abschnitt 4.6) vom Prozessor bestimmt werden muß, mit dem im Befehl stehenden Code übereinstimmt (CONDITION TRUE). Es kann also entsprechend jeder erfüllten Bedingung (siehe Bild 4.19 bzw. 4.5) ein Sprung programmiert werden.

Die dritte Gruppe von Sprungbefehlen ähnelt der zweiten; nur wird hier der Sprung ausgeführt, wenn die beiden Codes nicht übereinstimmen, d.h. die Bedingung nicht erfüllt ist (CONDITION FALSE). **2650**

4.9.3 Befehle für den Anruf von Unterprogrammen

Die Unterprogrammtechnik spielt bei der Programmierung aller Mikroprozessoren eine große Rolle. Alle Prozessoren haben Befehle, durch die der Einsprung in ein Unterprogramm erfolgt, und weitere Befehle für die Rückkehr aus dem Unterprogramm in das Hauptprogramm. Die Unterprogramm-Einsprungbefehle ähneln den normalen Sprungbefehlen mit einer wichtigen Ausnahme.

Beim Einsprung in ein Unterprogramm sorgt der Prozessor selbst dafür, ohne daß der Programmierer daran zu denken hat, daß die Adresse, die dem Sprungbefehl folgt, in geeigneter Weise abgespeichert wird, damit das Programm nach Beendigung des Unterprogramms dort wieder fortgesetzt werden kann.

Für das Abspeichern der Rücksprungadressen wird im allgemeinen ein Stapelspeicher, gelegentlich auch Kellerspeicher (STACK) genannt, verwendet. Organisation und Anordnung dieses Speichers sind allerdings bei den verschiedenen Mikroprozessoren unterschiedlich.

Wie im Bild 4.29 dargestellt, gibt es beim *6502* nur jeweils einen Einsprungbefehl für den Unterprogrammanruf und einen Rücksprungbefehl. Der Einsprungbefehl ist ein Drei-Byte-Befehl mit absoluter Adressierung, so daß die Unterprogramme in jedem beliebigen Teil des Adreßraums angeordnet werden können. Für den Rücksprungbefehl wird nur ein Byte benötigt. **6502**

Bei diesem Prozessor ist der Stapelspeicher grundsätzlich auf der Seite 1 des Speicherraums, beginnend mit der Adresse **01 FF**, angeordnet. Von dieser Adresse ausgehend, kann der Speicher mit abnehmender Adresse gefüllt werden. Der Stapelzeiger (SP = STACK POINTER) hat 8 Bit Breite und nimmt als Inhalt den unteren Teil der gerade gültigen Stapelspeicher-Adresse auf.

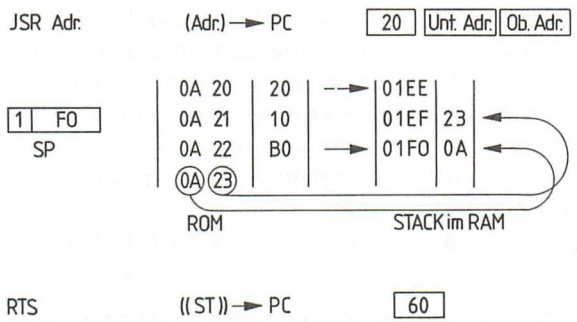


Bild 4.29 Einsprung- und Rücksprungbefehle für Unterprogramme beim 6502 mit dem Aufbau des Stapelspeichers (STACK) im RAM

In dem im Bild 4.29 dargestellten Beispiel wird durch den Befehl, der in der Adresse **0A 20** beginnt, ein Unterprogramm an der Adresse **B0 10** angerufen. Der Stapelzeiger hat den Inhalt **F0** und zeigt damit auf die Adresse **01 F0.** Dort wird der obere Teil der dem Einsprungbefehl folgenden Adresse, also **0A,** abgespeichert. Der untere Teil der Adresse, also **23,** kommt in die nächstniedrigere Speicherstelle, und nach Ausführung des Befehls zeigt dann der Stapelzeiger auf die Speicherstelle **01 EE.**

6502

6800

Der *6800* kennt für die Unterprogramm-Bedienung drei Einsprungbefehle und einen Rücksprungbefehl, die im Bild 4.30 dargestellt sind.

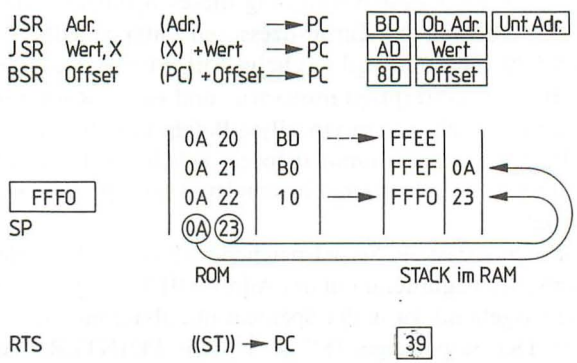


Bild 4.30 Einsprung- und Rücksprungbefehle für Unterprogramme beim 6800 mit dem Aufbau des Stapelspeichers (STACK) im RAM

Auch beim Unterprogramm-Anruf ist also absolute Adressierung, Index-Adressierung und relative Adressierung möglich.

Da der Stapelzeiger bei diesem Prozessor eine Kapazität von 16 Bit hat, kann der Stapelspeicher beliebig im Speicherraum gewählt werden. Es ist jedoch üblich, ihn an die höchstwertigen Adressen, die hardwaremäßig an das System angeschlossen sind, zu legen.

Im Prinzip arbeitet der Stapelspeicher genauso wie beim *6502* beschrieben, jedoch muß man bedenken, daß die Reihenfolge der Adreßteile beim *6800* anders ist. Der obere Adreßteil kommt auch immer in die höherwertige Adresse und der untere Adreßteil in die niederwertige Adresse.

Auch hier gibt es nur einen unbedingten Rücksprungbefehl mit 1 Byte Länge.

	3 Byte. Absolute Adressierung Unterprogrammanruf			1 Byte. Rückkehr v. U-Progr.	
Unbedingt	CALL	Adr.	CD	RET	C9
Zero = 0	CNC	Adr.	C4	RNZ	C0
Zero = 1	CZ	Adr.	CC	RZ	C8
Carry = 0	CNC	Adr.	D4	RNC	D0
Carry = 1	CC	Adr.	DC	RC	D8
Parity = 0	CPO	Adr.	E4	RPO	E0
Parity = 1	CPE	Adr.	EC	RPE	E8
Sign = 0	CP	Adr.	F4	RP	F0
Sign = 1	CM	Adr.	FC	RM	F8

Bild 4.31 Befehle für Einsprung und Rücksprung bei Unterprogrammen für den 8080

6800

8080

Abweichend von den anderen beiden Prozessoren hat der *8080* außer je einem unbedingten Einsprungbefehl und einem unbedingten Rücksprungbefehl noch eine ganze Anzahl von Befehlen, durch die Sprünge nur veranlaßt werden, wenn gewisse Bedingungen erfüllt sind. Die Sprungbedingungen werden, wie aus Bild 4.31 zu ersehen, wieder von dem Inhalt der Ergebnis-Bit im Programm-Status-Wort abgeleitet. Alle Einsprungbefehle sind Drei-Byte-Befehle mit absoluter Adressierung, während die Rücksprungbefehle auch hier nur 1 Byte Länge haben.

Unbedingte Sprünge zum Unterprogramm

BSTR, 3	Offset	(PC) ± Of → PC	3B	Offset		
BSTA, 3	Adr.	(Adr.) → PC	3F	Ob. Adr.	Unt. Adr.	
ZBSR	Offset	(H' 2000) ± Of → PC	BB	Offset		
BSXA	Adr.	(Adr.)+(R3) → PC	BF	Ob. Adr.	Unt. Adr.	

Sprung zum Unterprogramm wenn Register nicht Null							
Register →		R0	R1	R2	R3		
BSNR , R	Offset	78	79	7A	7A	(PC) ± Of	→ PC
BSNA , R	Adr.	7C	7D	7E	7F	(Adr.)	→ PC

Sprung zum Unterprogramm wenn Bedingungscode gleich						
Bedingungs-Code →		00	01	10		
BSTR , B	Offset	38	39	3A	(PC) ± Of	→ PC
BSTA , B	Adr.	3C	3D	3E	(Adr.)	→ PC

Sprung zum Unterprogramm wenn Bedingscode ungleich						
Bedingungs-Code →		00	01	10		
BSFR , B	Offset	B8	B9	BA	(PC) ± Of	→ PC
BSFA , B	Adr.	BC	BD	BE	(Adr.)	→ PC

RETC , 3	Unbedingter Rücksprung	17
RETE , 3	Unbed. Rücksprung und Freigabe Interrupt	37

Rücksprung wenn Bedingungscode gleich					
Bedingungs-Code →	00	01	10		
RETC , B	14	15	16	Rücksprung	
RETE , B	34	35	36	Rücksprung und Freigabe Interrupt	

Bild 4.32 Befehle für Einsprung und Rücksprung bei Unterprogrammen für den 2650

Die Organisation des Stapelspeichers beim *8080* wurde bereits im Abschnitt 4.8 behandelt und ist im Bild 4.25 dargestellt. **8080**

Die vielseitigste Möglichkeit für den Anruf von Unterprogrammen bietet wieder der *2650*. Der Befehlsaufbau und die Adressiermöglichkeiten stimmen praktisch mit den Sprungbefehlen, die im vorigen Abschnitt geschildert wurden, überein. **2650**

Wie aus Bild 4.32 zu ersehen, gibt es vier unbedingte Einsprungbefehle, mit denen ein Unterprogramm angerufen werden kann, und sechs bedingte Befehle, die genauso aufgebaut sind wie die einfachen Sprungbefehle.

Für den Rücksprung von einem Unterprogramm gibt es zwei unbedingte Rücksprungbefehle und zwei Befehle, bei denen der Rücksprung nur erfolgt, wenn die Bit im Bedingungscode gleich sind. Durch einen der Befehle wird jeweils die vorher festgelegte Sperre des Interrupts wieder aufgehoben.

Beim *2650* befindet sich der Stapelspeicher für die Rücksprung-Adressen direkt auf dem Prozessor-Chip. Maximal können bis zu 8 Adressen gespeichert werden, und dementsprechend hat der dazugehörige Stapelzeiger, der im oberen Programm-Status-Wort angeordnet ist, eine Breite von 3 Bit, worin die Zahlen von **0 bis 7** in binärer Form gespeichert werden können. Die Funktion dieses Stapelspeichers ist auf dem Bild 3.11 dargestellt. **2650**

4.10 Befehle zur Dezimalkorrektur

Grundsätzlich arbeiten die Rechenwerke in allen Mikroprozessoren nur rein binär, d.h., alle dort verarbeiteten Bitmuster werden als Binärzahlen interpretiert und entsprechend verarbeitet.

Da wir Menschen mit Binärzahlen nichts anfangen können, muß vor jedem Rechenvorgang eine Umcodierung von den uns geläufigen Dezimalzahlen in das Binärsystem und nach dem Rechenvorgang wiederum eine Umcodierung vom Binär- in das Dezimalsystem durch entsprechende Programmbefehle vorgenommen werden. Da dies für kleine Rechenaufgaben verhältnismäßig umständlich ist, können die meisten Mikroprozessoren zumindestens Additionen, oft auch Subtraktionen, auch im BCD-System ausführen.

Hierzu ist es notwendig, nach jedem Additions- bzw. Subtraktionsvorgang eine Dezimalkorrektur auszuführen, wie dies im Abschnitt 2.4 beschrieben wurde.

6502 Am einfachsten ist das Rechnen im BCD-System beim *6502*. Bei diesem Prozessor kann man durch Setzen des Bit Nr. 3 im Programm-Status-Wort festlegen, daß der Prozessor grundsätzlich «dezimal», d.h. im BCD-System rechnet. Die entsprechenden Korrekturen werden vom Prozessor selbständig durchgeführt.

6800 **8080** *6800* und *8080* kennen Ein-Byte-Befehle zur Dezimalkorrektur, die wie folgt lauten:

6800	DAA	Dezimalkorrektur Akku A	19
8080	DAA	Dezimalkorrektur Register A	27

Diese Prozessoren sind in der Lage, ebenso wie auch die anderen, *gepackte* (PACKED) *BCD-Zahlen,* worunter man 2 Zahlen pro Byte versteht, zu addieren.

Als Ergebnis entstehen auch wieder zwei Dezimalzahlen in einem Byte, wenn nach dem Additionsbefehl der Befehl zur Dezimalkorrektur gegeben wurde. Hierbei wird auch ein Übertrag richtig berücksichtigt.

Die Methode der zusätzlichen Addition des Wertes **6** oder **66,** die bei diesen Prozessoren verwendet wird, funktioniert allerdings nicht so einfach bei Subtraktionen im Zweier-Komplementsystem. Diese müssen, entsprechend dem Beispiel, das auch im Abschnitt 2.4 gegeben wurde, besonders programmiert werden.

2650 Der *2650* arbeitet darum mit einer anderen Art der Dezimalkorrektur, bei der durch die Korrekturbefehle

DAR	R0	R1	R2	R3
	94	95	96	97

je nach dem Inhalt des Carry und des Halb-Carry-Bit bei jedem Halb-Byte gegebenenfalls der Hexadezimalwert «**A**» hinzuaddiert wird, wofür die folgende Tabelle gilt:

Carry-Bit	0	I	0	I
Halb-Carry-Bit	0	0	I	I
Zusätzliche Addition	AA	0A	A0	00

Bei der Addition zur Dezimalkorrektur werden keine Übertrags-Bit verändert.

Bei dieser Art der Dezimalkorrektur ist es allerdings notwendig, vor Ausführung einer Addition zum Summanden den Hexadezimalwert **66** zu addieren. Dann folgt der eigentliche Additionsbefehl und danach der Befehl zur Dezimalkorrektur.

Bei Subtraktionen braucht nach dem Subtraktionsbefehl nur der Korrekturbefehl gegeben zu werden, um das Ergebnis richtig als zweistellige BCD-Zahl zu erhalten. Im Abschnitt 2.4 sind Beispiele zu finden, die dies erläutern. ***2650***

4.11 Sonstige Befehle

Bei den meisten Mikroprozessoren gibt es noch eine Anzahl von Kurzbefehlen, die nicht in das allgemeine Befehlsschema hineinpassen.

Jeder Prozessor kennt einen Befehl, bei dem nur der Befehls-Adreßzähler um den Wert **1** erhöht wird, während sonst keine Operation stattfindet. Die Befehlscodes für diesen NON-OPERATIONS-Befehl lauten wie folgt:

2650	NOP	Keine Operation	C0
6502	NOP	Keine Operation	EA
6800	NOP	Keine Operation	01
8080	NOP	Keine Operation	00

Auch gibt es bei allen Prozessoren einen Befehl, durch den die Arbeit des Prozessors angehalten wird, um z.B. auf einen Interrupt zu warten. Diese Befehle lauten wie folgt:

2650	HALT	40
6502	BRK	00
6800	WAI	3E
8080	HLT	76

Im Zusammenhang mit dem Interrupt-Vorgang gibt es beim 8080 bestimmte Befehle, um diesen Vorgang zu steuern.

EI FB gibt den Interrupt frei
DI F3 sperrt den Interrupt

4.12 Einrichtungen für den Programmstart

Alle Mikroprozessoren haben Einrichtungen, durch die sichergestellt wird, daß der Programmstart nach dem Einschalten an einer genau definierten Startadresse erfolgt. Weiterhin ist Vorsorge dafür getroffen, daß der Programmierer bestimmen kann, an welcher Stelle des Speicherbereichs der Prozessor nach einem Interrupt, also einer Programmunterbrechung, seine Arbeit fortsetzen soll.

Alle Prozessoren haben einen *Rücksetz-*(RESET-)*Eingang,* an den hardwaremäßig ein entsprechendes Startsignal gegeben werden kann. Wird dort nach dem Einschalten der Stromversorgung ein hohes Spannungsniveau angelegt, führt der Prozessor zunächst eine Reihe von Operationen aus, um den eigentlichen Programmbeginn vorzubereiten. Der Programmstart erfolgt dann entweder sofort oder wenn die Reset-Leitung wieder auf niedriges Spannungsniveau geht.

Beim *2650* und beim *8080* erfolgt der Programmstart nach einem Reset-Vorgang grundsätzlich von der Adresse **00 00** an.

Im Gegensatz hierzu muß die Startadresse, an der der Prozessor sein Programm an einem Reset-Vorgang beginnen soll, beim *6502* und beim *6800* im Programm programmiert werden. Der *6502* holt die Startadresse aus den Speicherstellen **FFFC** und **FFFD** und setzt die dort hinterlegten Daten als Adresse in den Befehlsadreß-Speicher für den Beginn des Programms. Die Adressen **FFFC** und **FFFD** dienen also als indirekter Zeiger.

Auch der *6800* arbeitet mit indirekten Adreßzeigern für den Programmstart. Hier werden meist die Speicherstellen **FFFE** und **FFFF** für die Ablage der Programmstartadresse verwendet.

Ein Mikroprozessor muß aber auch an einer genau definierten Adresse beginnen, wenn ein Interrupt erfolgt ist und das derzeit laufende Hauptprogramm zugunsten eines für die Interruptbedienung notwendigen Unterprogramms unterbrochen werden soll.

Beim *2650* und beim *8080* gibt es jeweils eine Interrupt-Anforderungsleitung. Wenn ein peripheres Gerät eine Programmunterbrechung anfordert und diese vom Prozessor angenommen wird, muß das Gerät eine Information auf den Datenbus geben, die vom Prozessor für die Ermittlung der Startadresse für das Interrupt-Serviceprogramm verwendet wird.

Beim *2650* wird ein Interrupt-Vektor verwendet, d.h. ein 8-Bit-Muster,

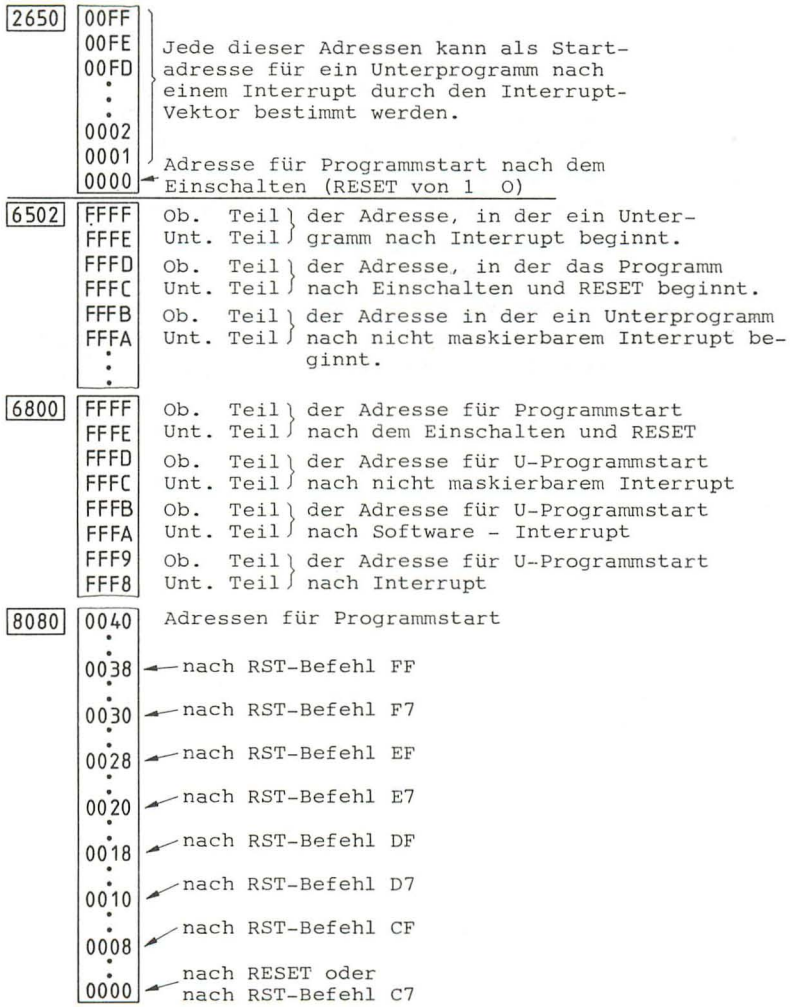


Bild 4.33 Bei den verschiedenen Prozessortypen vorgesehene Adressen für den Programmstart nach Stromeinschaltung und nach Unterbrechung durch Interrupt

das unmittelbar die Startadresse für das betreffende Unterprogramm in der Seite 0 des Adreßraums bestimmt.

Demgegenüber muß beim *8080* das Bitmuster eines **RST**-Befehls auf den Datenbus gegeben werden. Jeder dieser Befehle kennzeichnet eine ganz bestimmte Startadresse, von der aus der Prozessor dann das Interrupt-Unterprogramm beginnt.

Sowohl beim *6502* als auch beim *6800* gibt es mehrere durch Hardware-Befehle auszulösende Interrupt-Möglichkeiten. Im Adreßraum am Ende des adressierbaren Speicherbereichs sind darum mehrere Zeiger angeordnet, in die jeweils Startadressen für Unterprogramme, die durch die verschiedenen Interrupt-Programme angerufen werden sollen, abgespeichert werden können. Die verschiedenen Möglichkeiten sind im Bild 4.33 dargestellt.

5
Beispiele für Mikroprozessor-Programme (Lernprogramme)

Bei allen Problemlösungen durch einen Mikrocomputer wird man im Programm eine große Anzahl von einzelnen Strukturelementen feststellen, mit denen Teilaufgaben gelöst werden. Da ein Mikroprozessor rein sequentiell arbeitet, wird eine Teilaufgabe nach der anderen erledigt. Es ist darum zweckmäßig, ein Mikroprozessor-Programm entsprechend strukturiert, das heißt in Teilaufgaben untergliedert, aufzubauen.

Im wesentlichen gibt es *3 Strukturelemente* für den Programmaufbau, die im Bild 5.1 dargestellt sind.

Die einfachste Programmstruktur ist *die Folge* (SEQUENCE). Bei ihr wird ein Befehl nach dem anderen, wie sie in aufsteigender Adreßreihenfolge im Befehlsspeicher stehen, abgearbeitet.

Die Programmstruktur der Folge kommt natürlich in allen anderen Programmstrukturen auch vor. Es wäre an sich auch möglich, Programme, bei denen keine Entscheidungen notwendig sind, nur als Folge von Befehlen zu schreiben. Dies ist jedoch in der Praxis unwirtschaftlich, da bei derartigen Programmen sehr viel Speicherraum benötigt wird.

Man wird also überall dort, wo dies sinnvoll ist, ein anderes Strukturelement, *die Schleife* (LOOP), verwenden. Da ein Mikroprozessor bei der Datenverarbeitung nur ganz einfache Operationen durchführen kann, ist es oft notwendig, gewisse Operationen mehrfach zu wiederholen. Anstatt die gleichen Befehle immer wieder im Programm aufzuführen, programmiert man eine Schleife, die dann mehrfach durchlaufen wird.

Das Strukturelement der Schleife besteht zunächst aus einem Befehl, durch den der *Schleifenzähler* auf einen definierten Ausgangswert gesetzt

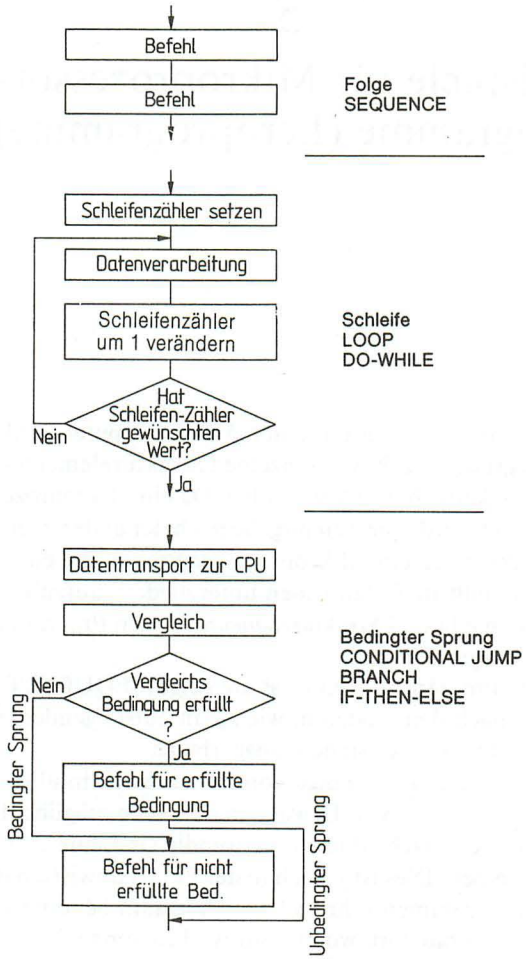


Bild 5.1 Die 3 Strukturelemente für den Programmaufbau

wird. Danach kommen der oder die Befehle, die die gewünschte Operation ausführen. Anschließend wird der Schleifenzähler um den Wert **1** verändert, was je nach Aufbau der Schleife eine Erhöhung oder Erniedrigung sein kann. Als nächstes kommt ein bedingter Sprungbefehl, der einen Rücksprung veranlaßt, wenn die vom Programmierer gesetzte Bedingung, ausgedrückt durch den Inhalt des Schleifenzählers, noch nicht erfüllt ist. Ist die Bedingung dagegen erfüllt, geht das Programm mit dem nächsten Befehl weiter.

Schleifen können auf verschiedene Weise aufgebaut werden. Man kann am Anfang den Schleifenzähler auf Null setzen und dann die Schleife abbrechen, wenn der Zähler zeigt, daß die gewünschte Anzahl von Schleifendurchgängen erfolgt ist. In diesem Fall ist es zweckmäßig, die notwendige Erhöhung des Schleifenzählers nach der Datenverarbeitung unmittelbar vor dem Sprungbefehl zu programmieren. Der Nachteil dieser Methode besteht darin, daß man meist noch einen besonderen Vergleichsbefehl braucht, um den Zählerstand abzufragen, bevor der Sprungbefehl ausgeführt werden kann.

Man arbeitet darum oft auch mit einer Decrementierung des Schleifenzählers und führt dann den Sprungbefehl solange aus, wie der Schleifenzähler noch nicht den Wert Null hat.

Man muß dabei allerdings richtig zählen, und es spielt eine Rolle, ob der Befehl, der den Schleifenzähler verändert, vor oder nach der Datenverarbeitung in die Programmfolge eingesetzt ist (siehe Bild 5.7).

Es kommt bei einem Mikroprozessor-Programm auch immer wieder vor, daß mehrere Schleifen ineinander verschachtelt werden. Man muß dann mehrere Schleifenzähler einrichten, was bei Prozessortypen, die mehrere CPU-Register aufweisen, besonders einfach ist.

Sehr wichtig für die Programmierung ist auch das Strukturelement *bedingter Sprung* (CONDITIONAL JUMP), das auch *Verzweigung* (BRANCH) genannt wird.

In diesem Strukturelement wird das Programm je nach dem Ergebnis einer *JA/NEIN-Entscheidung* auf verschiedene Weise fortgesetzt.

Als Kriterium für die Entscheidung gilt im allgemeinen, ob in einer bestimmten Bit-Position des Programm-Status-Worts eine **1** oder eine **0** steht. Gegebenenfalls muß vorher ein Vergleichsbefehl programmiert werden. Auf alle Fälle folgt dann ein bedingter Sprungbefehl, der auch ein Einsprungbefehl in ein Unterprogramm sein kann.

Wenn in einem Programm nicht nur eine JA/NEIN-Entscheidung

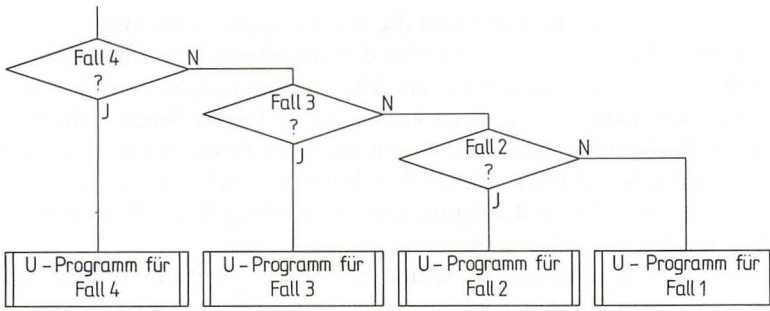


Bild 5.2 Fallabfrage für 4 mögliche Varianten

notwendig ist, sondern eine Reihe von Möglichkeiten, das Programm fortzuführen, bestehen soll, spricht man von einer *Fallabfrage*.

Programmtechnisch muß man aber jede Fallabfrage in einer Reihe von JA/NEIN-Entscheidungen auflösen, wie dies im Bild 5.2 gezeigt wird.

Die folgenden Programmbeispiele bestehen meist schon aus mehreren Strukturelementen. Es wird darin gezeigt, wie ein Mikroprozessor gewisse Aufgaben, die als Teilaufgaben einer größeren Problemstellung angesehen werden können, zu lösen vermag.

Für jede Aufgabenstellung, also für jedes Problem, gibt es im allgemeinen mehrere, oft sogar viele verschiedene Lösungsmöglichkeiten. Je nach dem internen Aufbau des Prozessors bzw. seinem Befehlssatz ergeben sich bei der Programmierung der verschiedenen Prozessoren oft auch verschieden aufgebaute Programme.

Für jede der in diesem Kapitel gestellten Programmieraufgaben werden vier verschiedene Lösungen mit den vier besprochenen Mikroprozessoren vorgestellt. Soweit möglich und sinnvoll, werden dabei bewußt verschiedene Lösungswege beschrieben, um die Vielfalt des Programmiermöglichkeiten, die Mikroprozessoren bieten, zu demonstrieren.

Um die Programme einigermaßen vergleichbar zu machen, werden sie alle mit der Adresse **1000** begonnen. Soweit Daten zu speichern sind, werden hierfür RAM-Speicherplätze von **0050** an aufwärts verwendet.

Eine gewisse Schwierigkeit bei der Auflistung von Programmen für verschiedene Mikroprozessoren stellen die Unterschiede in der Zahlendarstellung bei den verwendeten Assemblersprachen dar. In den Beispie-

len werden die im Bild 5.3 gezeigten Symbole, die von den Herstellern der Prozessoren auch verwendet werden, benutzt. Andere Besonderheiten der unterschiedlichen Syntax der Assemblersprachen werden jedoch in den Programmbeispielen nicht immer bis ins einzelne erklärt, da in diesem Buch in erster Linie die Programmierung in hexadezimaler Maschinensprache behandelt wird.

Es sei hier noch darauf hingewiesen, daß die folgenden Programme einer gewissen Ergänzung und Änderung bedürfen, wenn sie auf einem Mikroprozessor-Lern- und Übungsgerät ausprobiert werden sollen.

In jedem Fall sind die Befehlsadressen an die Adressen anzupassen, die im Gerät mit Speicherbausteinen bestückt sind. Soweit absolute Adressierung verwendet wird, müssen dann die betreffenden Befehle umgeschrieben werden.

Das gleiche gilt auch für programmierte Ein- und Ausgaben, die für die Port programmiert werden müssen, die in den betreffenden Geräten hardwaremäßig angeschlossen sind.

Wichtig ist es auch, daß alle Programme definiert beendet werden, da sonst zufällig nach dem Anwenderprogramm im Speicher stehende Werte Verwirrung stiften können. Zum Abschluß muß darum immer entweder

	Dez.-Zahlen	Bin.-Zahlen	Hex.-Zahlen	Index	Unmittelbar	Indirekt	Befehlszähler
2650 (Valvo)	Keine	B 'xx'	H'xx'	,R	im Op.-Code	*	$
8080 (Siemens, Intel)	Keine	xxx B	xxH	-	im Op.-Code	M	$
6502 (MOS, Rockwell)	Keine	%xx	$xx	,X ,Y	#	(Zeiger)	*
6800 (Motorola)	Keine	%xx	$xx	,X	#	-	*
9900 (Texas Instumets)	Keine		>xx	(R)	im Op.-Code	*	$

Bild 5.3 In den Assemblersprachen verschiedener Prozessoren verwendete Symbole

- ein Halt- oder Wartebefehl,
- ein Rücksprung zum Anfang des Programms oder
- ein Einsprungbefehl in das Monitorprogramm des betreffenden Geräts

in das Anwenderprogramm eingesetzt werden.

5.1 Programmbeispiel 1

Addition und Subtraktion von Binärzahlen

Lernziele:

Initialisieren. Arbeit mit und ohne Carry. Laden und Speichern. Benutzung von Additions- und Subtraktionsbefehlen. Setzen eines Merkers.

Aufgabe:

Drei Speicherstellen sollen mit positiven Binärzahlen geladen werden, von denen keine einen größeren Wert als dezimal 50 haben soll. Die Inhalte der drei Speicherstellen sind zu addieren, das Ergebnis ist abzuspeichern. Danach soll eine Kontrollrechnung durchgeführt werden und in einer Speicherstelle ein beliebiger Merker gesetzt werden, wenn die Addition richtig ausgeführt wurde.

Lösungsweg:

Da die Summe der drei Binärzahlen höchstens den Dezimalwert 150 betragen kann, wird für das Rechnen und Speichern des Ergebnisses höchstens ein Byte benötigt. Es kann also ohne Übertrag gerechnet werden.

Für die Kontrollrechnung werden Subtraktionsrechnungen verwendet. Wenn das Ergebnis Null ist, war die Addition richtig gerechnet.

Um das Programm möglichst einfach zu halten, wird der nach der Kontrollrechnung im Akkumulator stehende Wert **00** in den *Merker* ausgegeben. Gegebenenfalls kann durch einen einfacheren Befehl überprüft werden, ob in diesem Speicher der Inhalt **00** steht und damit die Additionsrechnung als richtig anzuerkennen ist.

Die Speicherstellen, in denen die zu addierenden Zahlen, das Ergebnis und der Kontrollmerker abzulegen sind, können vom Programmierer frei gewählt werden. Sie werden hier wie folgt bestimmt:

Symbolischer Name	ZAHL 1	ZAHL 2	ZAHL 3	SUMME	RICHTIG
RAM-Adresse	0050	0051	0052	005A	005B
Zahl im Beispiel	01	02	03	06	00

Als erstes soll hier das für den *6502* geschriebene Programm betrachtet werden, das ohne besondere Tricks aufgebaut ist (Programm 1.1). Da die Speicherstellen alle auf Seite Null des Adreßraums liegen, können sie bei diesem Prozessor mit Zwei-Byte-Befehlen angesprochen werden. **6502**

```
[1.1]                                  [6502]
 1 CLC                                  1000    18
 2 CLD                                  1001    D8
 3 LDA        #01                       1002    A901
 4 STA        $50                       1004    8550
 5 LDA        #02                       1006    A902
 6 STA        $51                       1008    8551
 7 LDA        #03                       100A    A903
 8 STA        $52                       100C    8552
 9 ADC        $51                       100E    6551
10 ADC        $50                       1010    6550
11 STA        $5A                       1012    855A
12 SEC                                  1014    38
13 SBC        $50                       1015    E550
14 SBC        $51                       1017    E551
15 SBC        $52                       1019    E552
16 STA        $5B                       101B    855B
```

Es wurde bereits gesagt, daß in diesem Programm das Carry-Bit nicht benötigt wird. Damit nicht eine dort zufällig im Prozessor stehende **1** das Programm verfälscht, wird der erste Befehl benutzt, um das Carry-Bit im Programm-Status-Wort auf **0** zu setzen.

Da hier rein binär gerechnet werden soll, der *6502* jedoch sowohl binäre als dezimale Arbeitsweise zuläßt, muß auch noch das Bit Nr. 3 im Programm-Status-Wort auf **0** gesetzt werden, was durch den zweiten Programmbefehl geschieht.

Hieraus ist schon gleich zu lernen, daß vor Beginn eines Programms jeder Prozessor in einen ganz bestimmten Ausgangszustand gebracht

werden muß, denn beim Stromeinschalten können alle Register einen nicht vorher bestimmbaren, d.h. rein zufälligen Inhalt annehmen. Man nennt diesen Vorgang *Initialisieren.*

Entsprechend der Aufgabenstellung wird jetzt zunächst die Zahl **01** unmittelbar in den Akkumulator geladen und danach der Inhalt des Akkumulators in die für die erste Zahl bestimmte Speicherstelle abgespeichert. Zur Vereinfachung der Kontrolle werden die Zahlen **01, 02** und **03** in diesem Programm verwendet. Entsprechend der Aufgabenstellung hätte natürlich jede andere Zahl, die den vorgegebenen Bereich nicht überschreitet, in die jeweils zweiten Befehls-Byte der Ladebefehle eingesetzt werden können.

In gleicher Weise wird anschließend die Zahl **02** und die Zahl **03** in den Akkumulator geladen und dann der Akkumulatorinhalt wieder abgespeichert. Nach dem achten Befehl befindet sich also noch die Zahl **03** im Akkumulator, der als Datenquelle durch den vorhergehenden Speicherbefehl seinen Inhalt nicht verloren hat.

Man erkennt, daß in der Assemblersprache dieses Prozessors das Dollarzeichen ($) immer dann steht, wenn Hexadezimalzahlen gemeint sind. Das Zeichen # kennzeichnet Befehle mit unmittelbarer Adressierung.

Jetzt kommt der erste Additionsbefehl mit Seite-Null-Adressierung, durch den der Inhalt der angesprochenen Speicherstelle zum Inhalt des Akkumulators addiert wird. Anschließend folgt der zweite Additionsbefehl, durch den noch die dritte Zahl zuaddiert wird. Damit ist die gewünschte Addition beendet, der Akkumulator enthält die Summe. Diese wird nun, wie beschrieben, auch wieder mit Seite-Null-Adressierung in die Speicherstelle **00 5A** abgelegt.

Jetzt könnte eigentlich die vorgesehene Kontrollsubtraktion beginnen. Es gibt aber eine Schwierigkeit. Im Carry-Bit des Programm-Status-Worts befindet sich eine **0,** die am Anfang des Programms eingeschrieben und durch die bisher ausgeführten Befehle nicht verändert wurde. Eine **0** im Carry-Bit bedeutet jedoch bei Subtraktionen bei diesem Prozessor, daß vorher geborgt wurde, was dann bei dem nachfolgenden Subtraktionsbefehl mit berücksichtigt wird. In diesem Fall ist dies aber nicht nötig, und so muß vor dem Subtraktionsbefehlen durch den Befehl **SEC** das Carry-Bit wieder auf **1** gesetzt werden.

Im Akkumulator steht immer noch die Summe, und so können jetzt

nacheinander mittels Seite-Null-Adressierung die in den Speicherstellen abgelegten Zahlen von der Summe subtrahiert werden.

Wenn die Additionen und selbstverständlich auch die Subtraktionen richtig durchgeführt worden sind, muß vor dem 16. Befehl als Ergebnis im Akkumulator **00** stehen. Dies ist der vereinbarte Wert im Merkerspeicher, der anzeigt, daß die Rechnung richtig war, und so braucht nur der Akkumulatorinhalt mit dem 16. Befehl in den Merkerspeicher ausgegeben zu werden.

Damit ist das Programm beendet. Gegebenenfalls kann jetzt durch einen weiteren Befehl abgefragt werden, ob im Merkerregister wirklich **00** steht; wenn nicht, kann man z.B. die Rechenvorgänge noch einmal wiederholen lassen.

6800

1.2

```
 1 LDAA    #1        1000    8601
 2 STAA    ZAHL 1    1002    B70050
 3 LDAA    #2        1005    8602
 4 STAA    ZAHL 2    1007    B70051
 5 LDAB    #3        100A    C603
 6 STAB    ZAHL 3    100C    F70052
 7 ABA               100F    1B
 8 ADDA    ZAHL 1    1010    BB0050
 9 STAA    SUMME     1013    B7005A
10 SBA               1016    10
11 SUBA    ZAHL 2    1017    B00051
12 SUBA    ZAHL 3    101A    B00052
13 STAA    RICHTIG   101D    B7005B
```

6800

Für den *6800* kann das gleiche Programm, wie im Programmbeispiel 1.2 dargestellt, geschrieben werden. Da dieser Prozessor verschiedene Befehle für Addition und Subtraktion mit und ohne Berücksichtigung des Carry-Bit hat, kann die Initialisierung entfallen.

Obwohl dieser Prozessor auch mit Seite-Null-Adressierung arbeiten kann, wird hier als Beispiel die absolute Adressierung angewandt. Im Assembler-Ausdruck werden die vorher festgelegten symbolischen Adressen verwendet.

Mit den ersten vier Befehlen werden nacheinander zwei Zahlen unmittelbar in den Akkumulator A geladen und von dort aus abgespeichert. Da

der Prozessor aber zwei Akkumulatoren besitzt, wird mit dem fünften Befehl die dritte Zahl in den Akkumulator B geladen und von hier aus abgespeichert.

Durch den siebten Befehl wird jetzt der Inhalt beider Akkumulatoren addiert, so daß durch den achten Befehl zu dem im Akkumulator befindlichen Ergebnis nur noch der Inhalt der Speicherstelle, die die Zahl 1 enthält, hinzuaddiert werden muß. Im neunten Befehl wird dann das Ergebnis der Addition abgespeichert, womit wiederum die Kontrollsubtraktion beginnen kann.

Da die Zahl 3 immer noch im Akkumulator B steht, wird zunächst durch den zehnten Befehl der Inhalt von B vom Inhalt des Akkumulators A abgezogen. Das Ergebnis bleibt in A. Dann kommen wieder zwei weitere Subtraktionsbefehle mit absoluter Adressierung, durch die zunächst die Zahl 2 und dann die Zahl 1 subtrahiert wird. Bei richtiger Rechnung muß der Akkumulator dann wieder das Ergebnis **00** aufweisen, das im Merkerspeicher abgelegt wird.

2650

1.3 — 2650

```
 1 CPSL     H'08'      1000   7508
 2 LODI,0   01         1002   0401
 3 LODI,1   02         1004   0502
 4 LODI,2   03         1006   0603
 5 STRA,0   H'0050'    1008   CC0050
 6 STRA,1   H'0051'    100B   CD0051
 7 STRA,2   H'0052'    100E   CE0052
 8 ADDZ,1              1011   81
 9 ADDZ,2              1012   82
10 STRA,0   H'005A'    1013   CC005A
11 SUBZ,1              1016   A0
12 SUBZ,2              1017   A1
13 SUBA,0   H'0050'    1018   AC0050
14 STRA,0   H'005B'    101B   CC005B
```

Das für den *2650* bestimmte Programm ist im Programmbeispiel 1.3 dargestellt. Dieser Prozessor muß wieder initialisiert werden. Durch den ersten Befehl wird das Bit Nr. 3 im unteren Programm-Status-Wort, das durch die Maske mit dem Hexadezimalwert **08** angesprochen wird, auf **0**

gesetzt und damit eine Arbeitsweise ohne Berücksichtigung des Carry programmiert.

Da der *2650* vier Universalregister in der CPU hat, werden die Register 0, 1, und 2 unmittelbar mit den drei Zahlen geladen und anschließend ihr Inhalt in die drei bestimmten Speicherstellen abgespeichert. Für die Additionen können die **ADDZ**-Kurzbefehle, bei denen das Register 0 jeweils den ersten Operanden und das Ergebnis aufnimmt, verwendet werden. Mit den **SUBZ**-Befehlen ist auch die Subtraktion sehr einfach.

Das Programm für den *8080*, um die gestellte Aufgabe zu lösen, kann auf verschiedene Weise geschrieben werden. In dem Programmbeispiel 1.4 wird eine Lösung, bei der vorwiegend das Registerpaar HL zur Adressierung benutzt wird, vorgestellt. Um das Programm besser verfol- **8080**

1.4 8080

				Inhalte	
				HL	A
1 LXI	0050H	1000	215000	0050	
2 MVI	M,ZAHL 1	1003	3601	0050	
3 INX	H	1005	23	0051	
4 MVI	M,ZAHL 2	1006	3602	0051	
5 INX	H	1008	23	0052	
6 MVI	M,ZAHL 3	1009	3603	0052	
7 MOV	A,M	100B	7E	0052	03
8 DCX	H	100C	2B	0051	03
9 ADD	M	100D	86	0051	05
10 DCX	H	100E	2B	0050	05
11 ADD	M	100F	86	0050	06
12 STA	SUMME	1010	325A00	0050	06
13 SUB	M	1013	96	0050	05
14 INX	H	1014	23	0051	05
15 SUB	M	1015	96	0051	03
16 INX	H	1016	23	0052	03
17 SUB	M	1017	96	0052	00
18 STA	RICHTIG	1018	325B00	0052	00

gen zu können, ist der Inhalt des Registerpaars HL und des Akkumulators nach jedem Befehl aufgeführt.

Eine Initialisierung ist auch bei diesem Prozessor für die gestellte Aufgabe nicht notwendig, da er unterschiedliche Befehle für arithmetische Operationen mit und ohne Carry besitzt.

Durch den ersten Befehl wird das Registerpaar HL mit der Adresse, in die die Zahl 1 abgespeichert werden soll, geladen. Es folgt dann ein Datentransport-Befehl, durch den die im zweiten Befehls-Byte stehende Information unmittelbar in die Speicheradresse, die im Register HL steht, abgespeichert wird.

Jetzt wird das Registerpaar HL um den Wert **1** erhöht und dann die nächste Zahl abgespeichert. Im fünften Befehl erfolgt eine weitere Incrementierung des Registerpaars; danach wird die Zahl 3 im Speicher abgelegt.

Durch den siebten Befehl wird jetzt der Inhalt der durch das Registerpaar HL adressierten Speicherstelle, das ist in diesem Fall die Zahl 3, in den Akkumulator transportiert. Durch den dann folgenden Befehl wird das Registerpaar HL wieder decrementiert und zeigt jetzt auf die Speicherstelle, in der die Zahl 2 abgelegt ist, so daß der Inhalt dieser Speicherstelle durch den an neunter Stelle folgenden Befehl zum Inhalt des Akkumulators addiert werden kann.

Durch die Befehle 10 und 11 wird der gleiche Vorgang noch einmal wiederholt und damit die Zahl 1 zum Akkumulatorinhalt hinzugefügt. Die Addition ist jetzt vollendet, der Akkumulator enthält die Summe. Sein Inhalt wird jetzt, durch den zwölften Befehl absolut adressiert, in die für die Summe vorgesehene Speicherstelle abgelegt.

Das Registerpaar HL zeigt immer noch auf die Speicherstelle, in der die Zahl 1 abgespeichert ist. Durch den dreizehnten Befehl wird deren Inhalt jetzt vom Akkumulator abgezogen. Danach wird das Registerpaar HL um den Wert **1** erhöht und eine weitere Subtraktion durchgeführt. Durch die Befehle 16 und 17 wird dieser Vorgang noch einmal wiederholt und als letztes der Inhalt der Speicherstelle, die die Zahl 3 enthält, vom Akkumulator subtrahiert. Jetzt ist auch die Kontrollrechnung beendet, und mit dem achtzehnten Befehl kann der Inhalt des Akkumulators in den Merkerspeicher mit der symbolischen Adresse **RICHTIG** transportiert werden.

5.2 Programmbeispiel 2

Verschachtelte Schleifen zur Zeitverzögerung

Lernziele:

Anruf von Unterprogrammen. Retten von Registerinhalten. Stapelspeicher-Bedienung. Parameter-Übergabe. Programmierung von Schleifen. Incrementieren und Decrementieren. Zeitberechnung. Symbolische Adressen. Berechnung von relativen Sprüngen.

Aufgabe:

Programmierung einer Zeitverzögerung von etwa 0,5 bis 50 Sekunden als jederzeit wieder verwendbares Unterprogramm.

Lösungsweg:

Bei der Programmierung eines Mikrocomputers ist es oft nützlich, größere oder kleinere Zeitverzögerungen durch Software-Befehle zu erzeugen und damit besondere Hardware-Zeitglieder einzusparen.

Da der Mikroprozessor einzelne Befehle in einer sehr kurzen Zeit ausführt, muß er für eine Zeitverzögerung veranlaßt werden, eine entsprechende meist sehr große Anzahl von Befehlsschritten zu durchlaufen, ohne daß dafür zu viele Programmbefehle notwendig sind. Man verwendet darum für diesen Zweck ineinander verschachtelte Schleifen, wie es im Flußschaubild 5.4 dargestellt ist.

Es werden hier drei Zählschleifen verwendet. Bei der inneren wird das mit R1 bezeichnete Register so lange decrementiert, bis es den Wert **00** erreicht hat. Es wird also immer wieder der gleiche Befehl angesprungen und der nächstfolgende Befehl erst dann verarbeitet, wenn die Entscheidung «hat das Register den Wert **00**» mit Ja beantwortet werden kann.

Dann wird zum erstenmal das Register 2 um den Wert **1** erniedrigt und auch hier gefragt, ob es schon den Wert **00** hat. Ist dies nicht der Fall, erfolgt ein Rücksprung. Das Register 1 wird neu geladen und wieder bis zum Wert **00** heruntergezählt.

Es ist zu erkennen, daß die innere Schleife, bei der das Register 1 beteiligt ist, sehr viel öfter durchlaufen wird als die mittlere Schleife mit dem Register 2.

Wenn die so mögliche Zeitverzögerung noch nicht ausreicht, kann

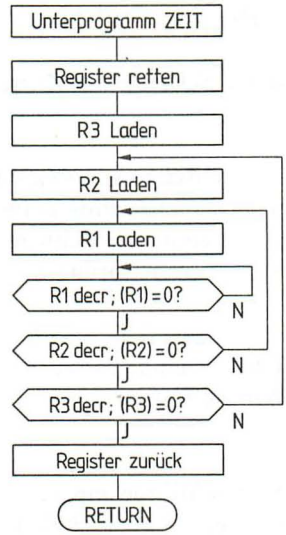


Bild 5.4 Aufbau eines Unterprogramms für Zeitverzögerung durch verschachtelte Zählschleifen

man noch ein drittes Register mit einer äußeren Schleife nachschalten, wie dies auch auf Bild 5.4 dargestellt ist.

Vor Beginn der eigentlichen Schleifen müssen die Register mit einem definierten Wert geladen werden, wenn man eine ganz bestimmte Zeitverzögerung erreichen will. Wie man die Verzögerungszeit ermittelt, wird bei einem der folgenden Beispiele gezeigt.

In der Aufgabenstellung wurde auch noch erwähnt, daß die Zeitverzögerung als *jederzeit wieder verwendbares Unterprogramm* programmiert werden soll. Man bezeichnet ein derartiges Programm als REENTRANT und versteht darunter ein Unterprogramm, das von verschiedenen Programmteilen aufgerufen werden kann und bei dem sichergestellt ist, daß bei seiner Abarbeitung nicht Daten oder andere Programmteile zerstört werden.

Es muß also dafür gesorgt werden, daß die Inhalte der CPU-Register, die vor dem Anruf des Unterprogramms bestanden, nicht durch das Unterprogramm zerstört werden. Man spricht hier vom *Retten der Registerinhalte* (SAVE), wofür es verschiedene Methoden gibt.

Natürlich ist bei der Programmierung auch die Forderung zu berücksichtigen, daß die Zeitverzögerung zwischen 0,5 und 50 Sekunden betragen soll. Wie lange das Programm die Zeit verzögern soll, muß also vor dem Aufruf als Zahl – man spricht hier von einem *Parameter* – in einer bestimmten Speicherstelle abgelegt werden, die dann durch das Unterprogramm übernommen wird. In diesem Fall handelt es sich um einen *Übernahmeparameter*. Ebenso kann es natürlich auch *Übergabeparameter* geben, wenn in einem Unterprogramm irgendein Wert ermittelt wird, der dann durch das Hauptprogramm weiterzuverarbeiten ist.

8080

Das Programmbeispiel 2.1 zeigt einen Programmiervorschlag für den *8080*. Als Speicherstelle für den Übergabeparameter wird hier die Adresse **00 60** verwendet. Da das Programm eine verhältnismäßig lange Zeitverzögerung ergeben soll, müssen drei ineinander verschachtelte Zeitschleifen verwendet werden, wobei bei den inneren Schleifen das Register immer einmal voll durchgezählt wird.

Für die äußere Schleife wird das Register A, für die mittlere Schleife das Register B und für die innere Schleife das Register C verwendet.

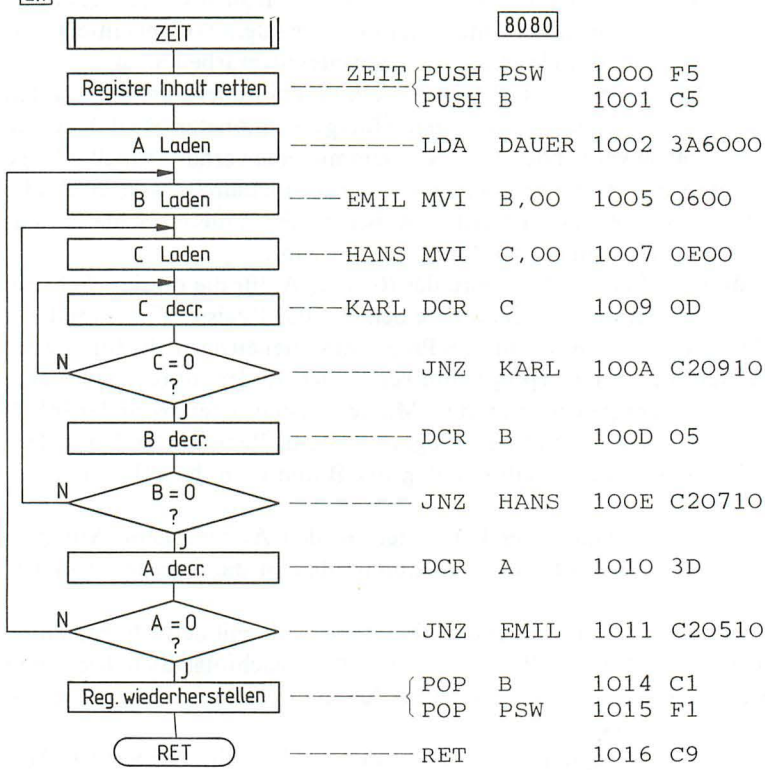
Die ersten beiden Befehle des Programms dienen dazu, die Inhalte der Register, die im Unterprogramm verwendet werden, durch Abspeicherung im Stapelspeicher zu retten. Mit dem ersten Befehl wird der Inhalt des Akkumulators und des Programm-Status-Registers und mit dem zweiten Befehl der Inhalt der Register B und C in den STACK übernommen.

Es folgt das Laden des Parameters in den Akkumulator. Mit dem Befehl 4 und 5 wird jeweils **00** unmittelbar in die Register B und C eingeschrieben.

Die Befehle 6 und 7 bilden die innere Schleife mit dem Decrementierungs-Befehl für das Register C und dem nachfolgenden bedingten Sprungbefehl, der immer dann auszuführen ist, wenn das Register nicht den Inhalt **00** hat.

Bei diesem Programmbeispiel werden symbolische Adressen **EMIL, HANS** und **KARL,** also vom Programmierer frei gewählte Ausdrücke, die jeweils einer ganz bestimmten Adresse entsprechen, verwendet. Vor dem Assemblieren des Programms, d.h., bevor das Programm in den Maschinencode übertragen wird, ist es notwendig, eine Liste der symbolischen Adressen aufzustellen. Wenn die Assemblierung durch einen Computer vorgenommen wird, geschieht dies im ersten Durchlauf des Assemblierprogramms. Wenn man ohne Computerhilfe arbeitet, muß

2.1


ZEIT
Register Inhalt retten
A Laden
B Laden
C Laden
C decr.
C = 0 ?
N
J
B decr.
B = 0 ?
N
J
A decr.
A = 0 ?
N
J
Reg. wiederherstellen
RET
8080
ZEIT PUSH PSW 1000 F5
PUSH B 1001 C5
LDA DAUER 1002 3A6000
EMIL MVI B,00 1005 0600
HANS MVI C,00 1007 0E00
KARL DCR C 1009 0D
JNZ KARL 100A C20910
DCR B 100D 05
JNZ HANS 100E C20710
DCR A 1010 3D
JNZ EMIL 1011 C20510
POP B 1014 C1
POP PSW 1015 F1
RET 1016 C9

man selbst eine Auflistung der symbolischen Adressen erstellen.
In diesem Fall bedeutet:

ZEIT	die Einsprung-Adresse für das Unterprogramm	**1000**
EMIL	die Adresse	**1005**
HANS	die Adresse	**1007**
KARL	die Adresse	**1009**
DAUER	die Adresse	**0060** für den Parameter

Man erkennt deutlich, daß für jede der drei Schleifen ein Ladebefehl, ein Decrementierbefehl und ein bedingter Sprungbefehl programmiert sind. Der Decrementierbefehl erfolgt vor dem Sprungbefehl, so daß jeweils beim ersten Schleifendurchgang der ursprünglich in das Register geladene Wert **00** zu **FF** decrementiert wird. Im Prinzip könnte man ein derartiges Programm natürlich auch mit Erhöhungs-, d.h. Incrementbefehlen, aufbauen, denn wenn der Registerinhalt **00** um **1** erhöht wird, erscheint dort auch **00** erst nach 256 Erhöhungen.

Zum Schluß des Programms müssen die im Stapelspeicher abgelegten Registerinhalte wieder in ihre alten Register transportiert werden, was durch die beiden **POP**-Befehle geschieht. Da der Stapelspeicher nach dem Prinzip *zuletzt EIN, zuerst AUS* arbeitet, muß der zuletzt dort abgespeicherte Inhalt als erstes wieder ausgelesen werden. Da das Register HL übrigens in diesem Unterprogramm nicht verwendet wurde, war es auch nicht notwendig, seinen Inhalt im Stapelspeicher abzuspeichern. Der Inhalt des Programmadreß-Registers wird ohnehin selbsttätig beim Anruf eines Unterprogramms dort abgelegt und wieder in das Befehlsadreß-Register transportiert, wenn als letztes im Unterprogramm ein Rücksprungbefehl erscheint.

2650

Für den *2650* ergibt sich bei Zeitverzögerungsschleifen ein besonders kurzes Programm, da bei diesem Prozessor ein Kombinationsbefehl für Decrementieren eines Registers und bedingten Sprung verwendet werden kann. Dieses Programm ist im Programmbeispiel 2.2 dargestellt.

Um die Inhalte der Register 1, 2 und 3 zu retten, bedarf es beim *2650*, der keinen allgemein zugänglichen Stapelspeicher kennt, nur eines Befehls zum Übergang auf die andere Registerbank. Die Register 1, 2 und 3 sind bekanntlich zweifach vorhanden. Wenn angenommen wird, daß im Hauptprogramm mit der Registerbank **0** gearbeitet wird, so kann

durch den Befehl **PPSL H'10'** der Übergang zur Registerbank **1** programmiert werden. Das Register 0 wird im Unterprogramm nicht verwendet, so daß auch sein Inhalt nicht zerstört wird.

Als zweiter, dritter und vierter Befehl kommen auch hier die Ladebefehle, wie schon im vorigen Beispiel erklärt. Für die drei Schleifen wird dann jeweils nur ein Befehl benötigt, da der **BDRR**-Befehl sowohl die Decrementierung des angesprochenen Registers als auch den Sprung veranlaßt, wenn dieses Register noch nicht **00** ist. Die im Programm verwendeten Sprungbefehle arbeiten mit relativer Adressierung.

2.2 2650

						Cycl	µs
1	ZEIT	PPSL	H'10'	1000	7710	3	9
2		LODA,3	DAUER	1002	0F006	3	9
3	EMIL	LODI,2	YY	1005	06YY	2	6
4	HANS	LODI,1	XX	1007	05XX	2	6
5	KARL	BDRR,1	KARL	1009	F97E	3	9
6		BDRR,2	HANS	100B	FA7A	3	9
7		BDRR,3	EMIL	100D	F876	3	9
8		CPSL	H'10'	100F	7510	3	9
9		RET,3		1011	17	3	9

Sprung Adresse \ Nächste Adresse	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
1	1	0	F	E	D	C	B	A	9	8	7	6	5	4	3	2
2	2	1	0	F	E	D	C	B	A	9	8	7	6	5	4	3
3	3	2	1	0	F	E	D	C	B	A	9	8	7	6	5	4
4	4	3	2	1	0	F	E	D	C	B	A	9	8	7	6	5
5	5	4	3	2	1	0	F	E	D	C	B	A	9	8	7	6
6	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8	7
7	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8
8	8	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9
9	9	8	7	6	5	4	3	2	1	0	F	E	D	C	B	A
A	A	9	8	7	6	5	4	3	2	1	0	F	E	D	C	B
B	B	A	9	8	7	6	5	4	3	2	1	0	F	E	D	C
C	C	B	A	9	8	7	6	5	4	3	2	1	0	F	E	D
D	D	C	B	A	9	8	7	6	5	4	3	2	1	0	F	E
E	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	F
F	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

Tabelle zur Berechnung relativer Sprünge

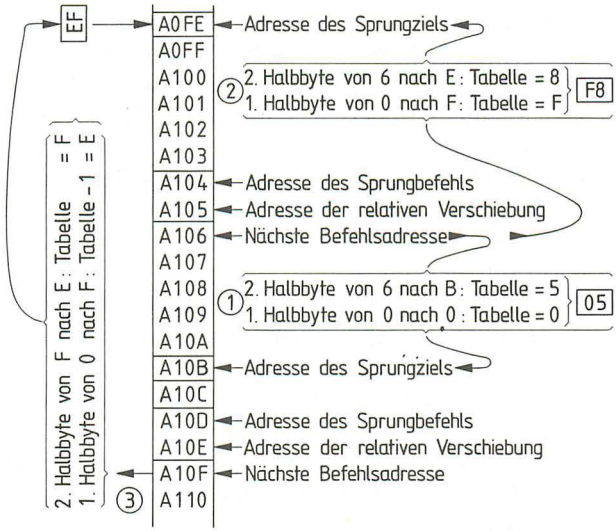


Bild 5.5 Tabelle zur Berechnung relativer Sprünge
Im Beispiel wird das Vorgehen bei Vorwärts- und Rückwärtssprüngen erläutert.

Bei (1) und (2) steht der Befehl mit relativer Adressierung in den Adressen **A104** und **A105.** Es soll nach **A10B** gesprungen werden. Die erste Ablesung in der Tabelle ergibt im Kreuzungspunkt von **6** und **B** den Wert **5,** der in das zweite Halb-Byte des zweiten Befehls-Byte zu schreiben ist. Die Ablesung lag unterhalb der Diagonale, so daß für das erste Halb-Byte keine Korrektur notwendig ist. Die Ablesung im Schnittpunkt von **0** und **0** ergibt **0** als Wert für das erste Halb-Byte.

In gleicher Weise ergibt sich für den Rücksprung aus der ersten Ablesung **8** aus der zweiten Ablesung **F.**

Für (3) ist die Adresse des Sprungzieles **A0FE;** der Befehl steht in **A10D** und **A10E.** Die erste Ablesung im Kreuzungspunkt von **F** und **E** liegt über der Diagonale. Darum muß die zweite Ablesung, die **F** ergibt, um **– 1** korrigiert werden, so daß in das erste Halb-Byte **E** einzusetzen ist.

Die Tabelle gilt für alle Prozessoren, bei denen der Bereich der relativen Verschiebung zwischen **– 128** und **+ 127** liegt. Ist der Bereich kleiner, muß ein Korrekturfaktor berücksichtigt werden. Beim *2850* muß z.B. berücksichtigt werden, ob der Befehl mit direkter oder indirekter Adressierung arbeiten soll. Es ist darum

- bei Vorwärtssprüngen mit indirekter Adressierung zum abgelesenen Wert hexadezimal **80** hinzuzuaddieren und
- bei Rückwärtssprüngen mit direkter Adressierung vom abgelesenen Wert hexadezimal **80** zu subtrahieren.

Bei diesem Programm wird mit relativer Adressierung gearbeitet. Die Berechnung der relativen Verschiebung bei der Programmierung in der Maschinensprache ist etwas schwierig, da der Mensch nicht gewöhnt ist, im Hexadezimal-Zahlensystem zu rechnen. Mit Hilfe der Tabelle in Bild 5.5 lassen sich relative Verschiebungen aber leicht bestimmen.

Ausgegangen wird bei der Benutzung der Tabelle einmal von der Adresse, die dem Befehl mit relativer Adressierung unmittelbar folgt. Zum zweiten muß noch die Zieladresse, also die Adresse, zu der der Sprung erfolgen soll, bekannt sein.

Man sucht zunächst oben in der Tabelle den Hexadezimalwert des letzten Adreß-Halb-Byte der nächsten Befehlsadresse und dann links den der Sprungadresse. Der in der Kreuzungslinie der beiden Spalten stehende Wert stellt das untere Halb-Byte der relativen Verschiebung dar.

In genau gleicher Weise geht man bei dem nächsthöheren Halb-Byte der Adresse vor. Im Kreuzungspunkt ist der Wert des oberen Halb-Byte der relativen Verschiebung zu finden. Dieser muß allerdings noch um den Wert **−1** korrigiert werden, wenn der Kreuzungspunkt der ersten Ablesung über dem Diagonalstrich in der Tabelle lag.

Am Schluß des Programms wird durch den **CPSL**-Befehl wieder die Registerbank **0** angerufen, gefolgt von einem unbedingten Rücksprungbefehl.

Anhand dieses Beispiels soll nun noch erklärt werden, wie man genaue Verzögerungszeiten programmieren kann. Aus den Programmtabellen der Hersteller ist zu entnehmen, wie viele Maschinenzyklen der Prozessor für die Ausführung des betreffenden Befehls benötigt. Aufgrund der Taktfrequenz und der inneren Struktur des Prozessors kann man dann ausrechnen, wie viele Mikrosekunden der einzelne Befehl beansprucht. Beides ist in diesem Beispiel neben den Befehlen vermerkt.

Nun muß man das Programm analysieren. Die ersten drei und die letzten beiden Befehle werden beim Anruf des Programms nur einmal durchlaufen. Die wenigen Mikrosekunden, die diese Befehle dauern, spielen an sich darum keine große Rolle. Jetzt ist zu ermitteln, wie lange der Durchlauf der einzelnen Zeitschleifen dauert. Wenn der Inhalt der Zeitschleifen-Register zunächst als **X, Y** und **Z** bezeichnet wird, ergibt sich die folgende Tabelle:

Takt = 1 MHz; 1 Cycle = 3 Takte = 3 µs

Reg. retten und laden (Bef. Nr. 1, 2, 8 und 9)		36
Innere Schleife 9 · X	X = 252	2 268
Mittlere Schleife (9 + 6 + inn. Schl.) · Y	Y = 220	500 060
Äußere Schleife (9 + 6 + mittl. Schl.) · (Z)	Z = 1	500 075

Inhalt in DAUER ≙ Z = 1 ≙ 0,500075 Verzögerungszeit.

Um die Programmierung der Zeitdauer mit Hilfe des Übergabeparameters möglichst einfach zu machen, sollte der Wert **1** im Parameter DAUER einem möglichst glatten Sekundenbruchteil entsprechen. Dies ist z.B. der Fall, wenn **X** gleich **252** und **Y** gleich **220** gewählt wird. Wenn dann **Z** gleich **1** gesetzt ist, dauert eine Zeitschleife **0,500075** Sekunden. Da im Programm keine ganz genaue Einhaltung der Zeiten gefordert wurde, dürfte dies hinreichend genau sein. Durch Verändern der beiden Werte **X** und **Y** und gegebenenfalls durch Hinzufügen von NOP-Befehlen, bei denen der Prozessor keine Operation ausführt, die aber Zeit verbrauchen, kann man die Zeit aber noch genauer festlegen. Es gibt auch entsprechende Formeln für diese Zeitberechnung, die aber auch nicht einfacher anzuwenden sind als die Probiermethode.

Wenn jetzt die Werte für **X** und **Y** als Dezimalzahlen ermittelt sind, müssen diese noch in Hexadezimal-Zahlen umcodiert werden, damit sie in die unmittelbaren Ladebefehle für die Register 1 und 2 eingesetzt werden können. Hierfür verwendet man eine Tabelle oder eines der im Kapitel 2 geschilderten Verfahren.

6502

Das Programmbeispiel 2.3 zeigt für den *6502* ein etwas anders aufgebautes Programm für drei ineinander verschachtelte Zeitschleifen. Da dieser Prozessor keine Hilfsregister auf dem Prozessor-Chip besitzt, müssen diese im RAM zweckmäßigerweise auf der Seite Null aufgebaut werden.

Im Programm wird zunächst der Inhalt des Akkumulators und der Inhalt des Programm-Status-Wortes im Stapelspeicher gerettet und dann mit dem dritten Befehl der Übernahmeparameter, der in der Speicherstelle **0060** mit der symbolischen Bezeichnung **DAUER** abgelegt wurde, in den Akkumulator geladen.

Als Hilfsregister werden die Speicherstellen **0050, 0051** und **0052** verwendet. In den vierten Befehl wird der Inhalt des Akkumulators und damit der Inhalt der Speicherstelle **DAUER** nach **0050** abgespeichert. Die

2.3			6502		
1		PHA		1000	48
2		PHP		1001	08
3		LDA	DAUER	1002	A560
4		STA	$0050	1004	8550
5	EMIL	LDA	#$YY	1006	A9YY
6		STA	$0051	1008	8551
7	HANS	LDA	#$XX	100A	A9XX
8		STA	$0052	100C	8552
9	KARL	DEC	$0052	100E	C652
10		BNE	KARL	1010	D0FC
11		DEC	$0051	1012	C651
12		BNE	HANS	1014	D0F4
13		DEC	$0050	1016	C650
14		BNE	EMIL	1018	D0EC
15		PLP		101A	28
16		PLA		101B	68
17		RTS		101C	60

Speicherstelle **DAUER** darf natürlich nicht selbst als Hilfsregister verwendet werden, da dann der dort abgespeicherte Wert beim nächsten Aufruf des Unterprogramms verloren wäre.

Mit dem fünften Befehl wird der Akkumulator dann mit dem Anfangswert für die mittlere Zählschleife geladen und dieser Wert anschließend nach **0051** abgespeichert. Dann wird der Akkumulator mit dem Anfangswert für den inneren Zählspeicher geladen, der dann auch diesmal wieder nach **0052** abgespeichert wird.

Mit dem neunten Befehl erfolgt die Decrementierung des Wertes im Speicher für die innere Zählschleife.

Diese Decrementierung wird so lange fortgesetzt, bis das Register den Wert **00** hat. Dann erfolgt zum erstenmal im elften Befehl die Decrementierung der mittleren Zählschleife usw. Da der *6502* die Decrementierung von Werten, die in Speichern stehen, durch einen einzigen Befehl zuläßt, wird das Programm auch bei diesem Prozessor, der keine Hilfsregister kennt, nicht wesentlich länger als bei anderen Typen.

Am Ende des Programms muß nun der alte Status wieder hergestellt werden. Durch die Befehle **PLP** und **PLA** werden Programm-Status-

Register und Akkumulator wieder mit dem alten Inhalt geladen. Am Schluß folgt dann der Rücksprungbefehl.

Zu erwähnen ist noch, daß man bei diesem Prozessor auch die Indexregister X und Y hätte als Zählregister verwenden können. Dann hätte man vorher den alten Inhalt dieser Register auch retten müssen, wozu jeweils zwei Befehle notwendig sind. Zunächst wird der Inhalt eines Indexregisters durch einen Transferbefehl **TXA** in den Akkumulator überschrieben und dann der Akkumulatorinhalt wiederum durch einen **PHA**-Befehl in den Stapelspeicher abgelegt. Umgekehrt muß am Ende des Programms verfahren werden.

6800

Wieder etwas anders kann das Zeitverzögerungsprogramm, wie das Programmbeispiel 2.4 zeigt, für den *6800* aufgebaut werden. Dieser Prozessor besitzt ein Indexregister mit 16 Bit Breite. Die Decrementierung dieses einen Registers dauert also genausolange wie die Decrementierung von 2 ineinander verschachtelten Registern bei den anderen Prozessoren. Man braucht hier also nur zwei Zeitschleifen, da die innere und mittlere Schleife zusammengefaßt sind.

2.4			6800		
1		PSHB		1000	37
2		STX	SAVE	1001	DF50
3		LDAB	DAUER	1003	9660
4	EMIL	LDX	#0000	1005	CE0000
5	KARL	DEX		1008	09
6		BNE	KARL	1009	26FD
7		DECB		100B	5A
8		BNE	EMIL	100C	26F7
9		PULB		100E	33
10		LDX	SAVE	100F	DE50
11		RTS		1011	39

Im Beispiel wird der Akkumulator B und das Indexregister benötigt; der Inhalt des Akkumulators B wird durch den Befehl **PSHB** in den Stapelspeicher eingeschrieben. Für die Rettung des Inhalts des Indexregisters werden die Speicherstellen **0050** und **0051** verwendet, die die symbolische Adresse **SAVE** erhalten. Durch den **STX**-Befehl wird der Inhalt des Indexregisters in 2 aufeinanderfolgende Speicherstellen abgespeichert.

Decrement- und bedingte Sprungbefehle arbeiten genauso wie bei den anderen Prozessoren beschrieben. Nach dem Zurückladen des alten Inhalts in den Akkumulator B wird auch der in der Speicherstelle **SAVE** stehende Inhalt wieder in das Indexregister zurückgeladen. Da Stapelspeicher und die Speicherstelle **SAVE** keine Beziehung zueinander haben, braucht in diesem Fall das Prinzip «zuerst EIN – zuletzt AUS» nicht beachtet zu werden, was bei der Datenrettung im Stapelspeicher immer notwendig ist.

5.3 Programmbeispiel 3

Der Größe nach sortieren

Lernziele:

JA/NEIN-Entscheidungen. Logische Vergleiche, Bedingte Sprünge. Anwenden von Vergleichsbefehlen, Aufbau eines Hilfsregisters. Bedingte und ewige Schleifen.

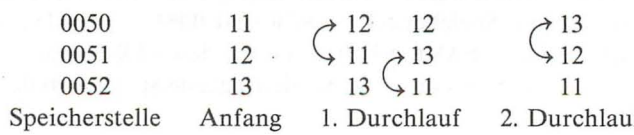
Aufgabe:

Die in den Speicherstellen mit den Adressen **0050, 0051 und 0052** *abgespeicherten Binärzahlen* **Z1, Z2** *und* **Z3** *sollen entsprechend der Größe ihres Zahlenwerts so sortiert werden, daß die größte Zahl in die Speicherstelle* **0050** *kommt.*

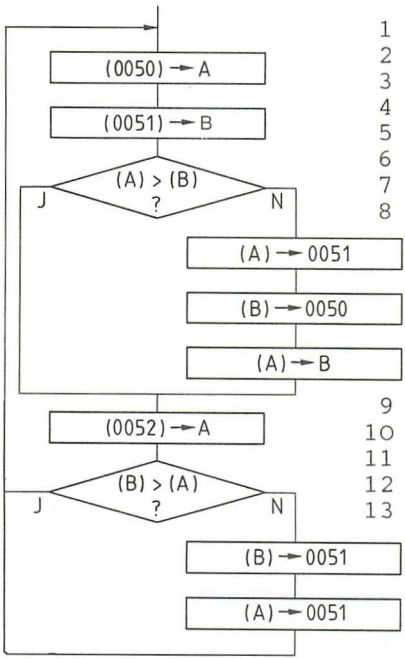
Lösungsweg:

Das Flußdiagramm zeigt die Verwendung von zwei Registern, deren Inhalt miteinander verglichen wird. Je nachdem, ob der Inhalt schon in richtiger Reihenfolge steht oder nicht, wird der Inhalt der Speicherstellen unverändert gelassen oder ausgetauscht.

Im zweiten Programmteil wird dann der Vergleich für den Inhalt der zweiten und dritten Speicherstelle ausgeführt. Gegebenenfalls muß das ganze Programm zweimal durchlaufen werden, wie das folgende Beispiel mit den Zahlen **11, 12** und **13** zu Beginn des Programms zeigt.



3.1



6800

```
 1 INGE LDAA 0050 A000 9650
 2      LDAB 0051 A002 D651
 3      CBA       A004 11
 4      BHI  ELSE A005 2205
 5      STAA 0051 A007 9751
 6      STAB 0050 A009 D750
 7      TAB       A00B 16
 8 ELSE LDAA 0052 A00C 9652
 9      CBA       A00E 11
10      BLS  INGE A00F 23EF
11      STAA 0051 A011 9751
12      STAB 0052 A013 D752
13      BRA  INGE A015 20E9
```

6800

Wenn das Sortierprogramm auf dem *6800* laufen soll, werden zweckmäßigerweise die beiden Akkumulatoren A und B zunächst mit dem Inhalt der Speicherstellen **0050** und **0051** geladen. Es folgt im dritten Befehl der Vergleich der Inhalte der beiden Akkumulatoren. Wenn der Inhalt des Akkumulators A größer ist, erfolgt der Sprung in den zweiten Programmteil, beginnend mit der symbolischen Adresse **ELSE.**

Ist der Inhalt von A nicht größer als der von B, wird A in die zweite Speicherstelle **0051** und B in die erste Speicherstelle **0050** abgespeichert. Der ursprüngliche Inhalt von A, also die Zahl, die als kleinere der beiden festgestellt wurde, wird jetzt nach B transferiert und der Akkumulator A mit der dritten Zahl aus der Speicherstelle **0052** geladen.

Es erfolgt erneut ein Vergleich zwischen den Zahlen, die in A und B stehen, wobei diesmal ein Vertauschen notwendig ist, wenn die in B stehende Zahl, also die Zahl, die zuletzt in der Speicherstelle **0051** stand, kleiner ist als die in A stehende Zahl aus der Speicherstelle **0052.**

Ist kein Austausch notwendig, springt das Programm zum Anfang und beginnt einen zweiten Programmlauf, in dem auch im ungünstigsten Fall die Sortierung von 3 Zahlen der Größe nach beendet wird.

6502

3.2 6502

```
 1 INGE   LDA    $50      1000   A550
 2        CPM    $51      1002   C551
 3        BPL    ELSE     1004   100A
 4        STA    $53      1006   8553
 5        LDA    $51      1008   A551
 6        STA    $50      100A   8550
 7        LDA    $53      100C   A553
 8        STA    $51      100E   8551
 9 ELSE   LDA    $51      1010   A551
10        CMP    $52      1012   C552
11        BPL    INGE     1014   10EA
12        STA    $53      1016   8553
13        LDA    $52      1018   A552
14        STA    $51      101A   8551
15        LDA    $53      101C   A553
16        STA    $52      101E   8552
17        JMP    INGE     1020   4C0010
```

Das gleiche Sortierprogramm wird im Beispiel 3.2 für den *6502* gezeigt. Es werden die gleichen Speicherstellen verwendet wie beim Beispiel 3.1; grundsätzlich wird mit Seite-Null-Adressierung gearbeitet. Bei diesem Prozessor gibt es Vergleichsbefehle, mit denen der Inhalt einer Speicherstelle mit dem Inhalt des Akkumulators verglichen werden kann.

Der zweite Befehl führt den Vergleich aus und ermittelt, ob der Inhalt des Akkumulators, in dem die Zahl aus der Speicherstelle **0050** steht, oder der Inhalt der Speicherstelle **0051** größer ist. Ist der Akkumulatorinhalt größer, braucht nicht umsortiert zu werden, und es folgt der Sprung in den zweiten Programmteil. Ist der Inhalt des Akkumulators kleiner, muß

umsortiert werden, was durch die Befehle 4 bis 8 geschieht. Für das Austauschen der Speicherinhalte muß noch die Hilfsspeicherstelle **0053** mit hinzugenommen werden, denn alle Datentransportvorgänge müssen bei diesem Prozessor über den Akkumulator laufen. Ohne Hinzunahme dieser Hilfsspeicherstelle würde der Inhalt einer der ursprünglichen Speicherstellen durch Überschreiben verlorengehen.

Im zweiten Teil des Programms von der Adresse **ELSE** an erfolgt der Vergleich der Zahlen, die in den Speicherstellen **0051** und **0052** gespeichert waren, nach dem gleichen Prinzip.

Die bedingten Sprungbefehle innerhalb des Programms arbeiten mit relativer Adressierung. Die Adressenrechnung kann mit Hilfe der Tabelle im Anhang kontrolliert werden.

Für den unbedingten Rücksprungbefehl am Schluß des Programms muß die absolute Adressierung Verwendung finden, wobei bei diesem Prozessor im zweiten Befehls-Byte zunächst der untere Adreßteil und dann im dritten Befehls-Byte der obere Adreßteil steht.

3.3 2650 **2650**

```
 1          PPSL   H'02'    1000   7702
 2 INGE     LODA,0 H'0050'  1002   0C0050
 3          LODA,1 H'0051'  1005   0D0051
 4          COMZ,1         1008   E1
 5          BCTR,1 ELSE    1009   1907
 6          STRA,0 H'0051'  100B   CC0051
 7          STRA,1 H'0050'  100E   CD0050
 8          STRZ,1         1011   C1
 9 ELSE     LODA,0 H'0052'  1012   0C0052
10          COMZ,1         1015   E1
11          BCTR,2 INGE    1016   1A6A
12          STRA,1 H'0052'  1018   CD0052
13          STRA,0 H'0051'  101B   CC0051
14          BCTR,3 INGE    101E   1B62
```

Im Beispiel 3.3 steht das gleiche Programm für den *2650* beschrieben. Da der Prozessor sowohl logische als auch arithmetische Vergleiche durchführen kann, wird hier zunächst das Bit Nr. 1 im PSU durch den Befehl **PPSL H'02'** auf **1** gesetzt. Dies bedeutet, daß der Prozessor mit

3.4 8080

						Registerinhalte 1. Lauf		2. Lauf	
						HL	A	B	A	B
1	INGE	LXI	H,0050H	1000	215000	0050				
2		MOV	A,M	1003	7E	0050	11		12	
3		INX	H	1004	23	0051	11		12	
4		MOV	B,M	1005	46	0051	11	12	12	13
5		CMP	B	1006	B8	0051	11	12	12	13
6		JP	ELSE	1007	F20E10	0051	11	12	12	13
7		MOV	M,A	100A	77	0051	11	12	12	13
8		DCX	H	100B	2B	0050	11	12	12	13
9		MOV	M,B	100C	70	0050	11	12	12	13
10		INX	H	100D	23	0051	11	12	12	13
11	ELSE	INX	H	100E	23	0052	11	12	12	13
12		MOV	B,A	100F	47	0052	11	11	12	12
13		MOV	A,M	1010	7E	0052	13	11	11	12
14		CMP	B	1011	B8	0052	13	11		
15		JM	INGE	1012	FA0010	0052	13	11	INGE	
16		MOV	M,B	1015	70	0052	13	11		
17		DCX	H	1016	2B	0051	13	11		
18		MOV	M,A	1017	77	0051	13	11		
19		JMP	INGE	1018	C30010	0051	13	11		

logischem Vergleich arbeitet, also das Bitmuster nicht als Zahl im Zweier-Komplementsystem ansieht. Bei arithmetischem Vergleich würde er dies tun und das Bit Nr. 7 nur als Vorzeichen behandeln.

Hier werden die beiden Zahlen in die Register 0 und 1 geladen und dann miteinander verglichen. Bei dem anschließenden Sprungbefehl **BCTR,1** erfolgt der Sprung, wenn der Bedingungscode, der durch den vorherigen Vergleichsbefehl gesetzt wurde, der Bedingung im Sprungbefehl entspricht. Die Bedingung **01** bedeutet, daß der Inhalt des Registers 0 größer als der Inhalt des Registers 1 ist. Ist dies nicht der Fall, muß wieder umsortiert werden; der Inhalt des Registers 0 muß durch den achten Befehl **STRZ,1** in das Register 1 transportiert werden.

Im zweiten Teil wird dann das Register 0 mit dem Inhalt der Speicherstelle **0052** geladen und wieder ein Vergleich durchgeführt. Veranlaßt durch den elften Befehl **BCTR,2** erfolgt in diesem Fall jedoch der Sprung zum Programmbeginn, d.h. also ohne Umsortieren, wenn die Bedingung **02** erfüllt ist, d.h. der Inhalt des Registers 0 kleiner ist als der Inhalt des Registers 1.

Das für den *8080* im Beispiel 3.4 geschriebene Programm kann man anhand der Hilfstabelle, die die Registerinhalte im ersten und zweiten Lauf zeigt, wenn die Anfangszahlen **11, 12** und **13** waren, leichter verfolgen. ***8080***

Durch den ersten Befehl wird zunächst das Registerpaar HL mit der Speicheradresse für die erste Zahl geladen. Unter Benutzung des Inhalts dieses Registerpaars als Zeiger wird dann im zweiten Befehl der Inhalt der Speicherstelle **0050** in den Akkumulator transportiert. Danach wird das Registerpaar HL incrementiert; es zeigt nun auf die Speicherstelle **0051,** deren Inhalt mit dem vierten Befehl in das Register B transportiert wird. Als fünftes folgt dann wieder ein Vergleichsbefehl. Der folgende Sprungbefehl wird ausgeführt, wenn der Inhalt von A größer ist als der von B.

Ist dies nicht der Fall, so erfolgt wieder eine Umsortierung. Da das Registerpaar HL immer noch auf die Adresse **0051** zeigt, kann der Inhalt des Akkumulators durch den siebenten Befehl **MOV M,A** dorthin abgespeichert werden.

Im achten Befehl wird dann das Registerpaar HL decrementiert, es zeigt nun auf die Speicherstelle **0050,** so daß mit dem neunten Befehl dorthin der Inhalt des Registers B abgelegt werden kann.

Jetzt wird das Registerpaar HL zweimal incrementiert, es zeigt dann

auf die Speicherstelle **0052,** deren Inhalt in das Register B transportiert wird.

Ist der Inhalt von A größer als der von B, d.h., stehen die Zahlen bereits in richtiger Reihenfolge, erfolgt der Sprung zum Beginn des Programms. Ist dies nicht der Fall, wird wieder umsortiert.

5.4 Programmbeispiel 4

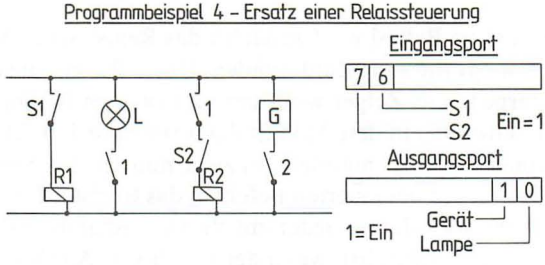
Ersatz einer Relaissteuerung durch Programm

Lernziele:

Festlegen von Ein- und Ausgabeport. Baustein-Initialisierung. Anwenden von Test-Befehlen. Prüfen von Teilen eines Bitmusters. Anwenden von Ein- und Ausgabebefehlen.

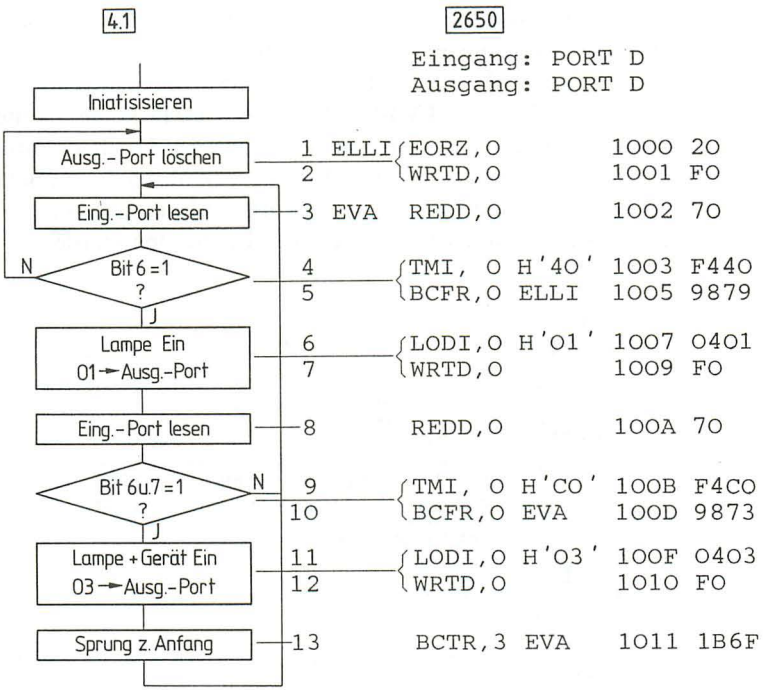
Aufgabe:

Die in der Zeichnung dargestellte Relaissteuerung soll durch eine Mikroprozessorsteuerung ersetzt werden.



Lösungsweg:

Zunächst ist der Schalter **1** so lange abzufragen, bis festgestellt wird, daß er eingeschaltet wurde. So lange er nicht eingeschaltet ist, wird die Lampe und das Gerät ausgeschaltet. Ist er eingeschaltet, wird die Lampe eingeschaltet und als zweites überprüft, ob Schalter **1** und Schalter **2** eingeschaltet sind. Ist dies der Fall, wird das Gerät eingeschaltet. Ist dies nicht der Fall, wird erneut überprüft, ob der Schalter **1** noch eingeschaltet ist, und gegebenenfalls Lampe und Gerät ausgeschaltet.



Das Beispiel 4.1 zeigt das Programm für den *2650*. Zunächst muß ein Eingangsport und ein Ausgangsport festgelegt und bestimmt werden, an welche Leitungen die beiden Schalter, die Lampe und das Gerät anzuschließen sind. Im Beispiel wird das **Port D** als Eingang und darin die Bit Nr. 7 und Nr. 6 für die Schalter **1** und **2** gewählt. Als Ausgang dient das **Port D,** die Lampe wird dann an das Bit Nr. 0 und das Gerät an das Bit Nr. 1 angeschlossen. Eine **1** im Bitmuster soll jeweils «eingeschaltet» bedeuten, was beim Anschluß der Hardware zu berücksichtigen ist.

Der *2650* braucht für das Port D keine besondere Initialisierung, so daß das Programm mit einem **EORZ**-Befehl beginnt, mit dem das Register 0 auf den Wert **00** gesetzt wird. Dies ist ein kleiner Programmiertrick, denn durch Exklusiv-ODER-Verknüpfung eines Bit-Musters mit sich selbst wird es immer auf **00** gestellt.

Mit dem zweiten Befehl wird der Inhalt des Register 0 an das Port

ausgegeben, was ein Ausschalten der Lampe und des Geräts bewirkt. Es folgt das Einlesen des Eingangsport in das Register 0.

Im vierten Befehl wird geprüft, ob das Bit Nr. 6 schon den Wert 1 hat. Hierfür wird beim *2650* der **TMI**-Befehl mit einer Maske als zweitem Befehls-Byte verwendet, wie es im Bild 5.6 erklärt ist. Die Bedingungscode-Bit im unteren Programm-Status-Wort werden durch diesen Befehl immer dann auf **00** gesetzt, wenn überall dort, wo in der Maske eine **1** steht, auch im zu prüfenden Bit-Muster eine **1** gefunden wurde.

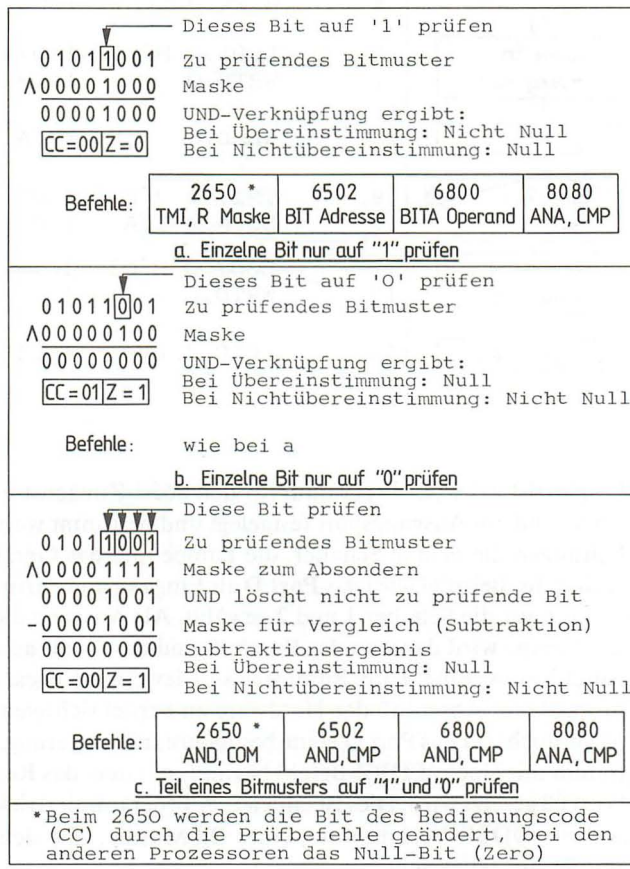


Bild 5.6 Methoden für das Prüfen einzelner Bit in einem Bit-Muster

Der als Befehl 5 folgende Sprungbefehl **BCFR,0** führt den Sprung aus, wenn keine Übereinstimmung zwischen Bedingungscode im Befehl und dem Bedingungscode, der durch den vorherigen Testbefehl in das Programm-Status-Wort eingeschrieben wurde, besteht. Das heißt also, der Sprung wird ausgeführt, solange das Bit Nr. 6 im Eingangsport noch nicht **1** ist. Besteht dagegen Übereinstimmung der beiden Bedingungscodes, so geht das Programm zum nächsten Befehl über, und es wird der Wert **01** zunächst in das Register 0 geladen und dann an den Ausgang ausgegeben, womit die Lampe eingeschaltet wird.

Anschließend wird der Inhalt des Eingangsport wieder in das Register 0 übertragen und ein erneuter Testbefehl, diesmal mit einer Maske, die in den beiden höchsten Bitpositionen eine **1** enthält, ausgeführt. Es erfolgt wieder ein bedingter Sprungbefehl, der ausgeführt wird, wenn keine Übereinstimmung der Bedingungscodes im Programm-Status-Wort und im Befehl festgestellt wird.

Besteht dagegen Übereinstimmung, wird der Wert **3** an den Ausgang ausgegeben, wodurch das Bit Nr. 0 und Nr. 1 auf **1** gesetzt wird, was die Lampe und das Gerät einschaltet.

Durch den unbedingten Sprungbefehl am Schluß des Programms werden beide Abfrageschleifen erneut durchlaufen. Es ist zu prüfen, ob inzwischen ein Schalter ausgeschaltet wurde.

6800

Im Prinzip ist das im Beispiel 4.2 für den *6800* dargestellte Programm gleich aufgebaut. Es wird jedoch angenommen, das eine **PIA 6820** als Ein-/Ausgabeeinheit Verwendung findet, die an die Adressen **8004** bis **8007** angeschlossen ist und die vor Programmbeginn programmiert werden muß.

Bei diesem Baustein, der zwei Port mit den Bezeichnungen A und B bedient, muß zunächst programmiert werden, ob durch den folgenden Befehl das eigentliche Port oder das Datenrichtungs-Register angesprochen wird. Hierzu werden die Kontrollregister mit den symbolischen Adressen **CONA** und **CONB** in den Bit-Positionen Nr. 3 entweder mit einer **0,** wenn das Datenrichtungs-Register gemeint ist, oder mit einer **1,** wenn das Ein-/Ausgabeport angesprochen werden soll, geladen.

Im Datenrichtungs-Register, das bei diesem Baustein die gleiche Adresse hat wie das Port, bestimmt eine **1** in einem Bit, daß diese zugehörige Leitung im Port als Ausgang und eine **0,** daß die Leitung als Eingang geschaltet ist.

4.2 6800

```
                PIA 6820      8004 . . . 8007

Ausgang A       PORTA EQU     $8004    1 = Ausgang
Eingang B       PORTE EQU     $8006    0 = Eingang
Kont.-Reg. A    CON A  EQU    $8005  } Bit 3 = 0:Richtung
Kont.-Reg. B    CON B  EQU    $8007  } Bit 3 = 1:Ein/Aus

 1          CLR    CONA      1000   7F8005
 2          LDAA   #$FF      1003   86FF
 3          STAA   PORTA     1005   B78004
 4          LDAA   #$04      1008   8604
 5          STAA   CONA      100A   B78005
 6          CLR    CONB      100D   7F8007
 7          CLR    PORTE     1010   7F8005
 8          STAA   CONB      1013   B78007
 9  ELLI    CLRA             1015   4F
10          STAA   PORTA     1016   B78004
11  EVA     LDAA   #$40      1019   B740
12          BITA   PORTE     101B   B58006
13          BEQ    ELLI      101E   27F5
14          LDAA   #$01      1020   8601
15          STAA   PORTA     1022   B78004
16          LDAA   #$80      1025   8680
17          BITA   PORTE     1027   B58006
18          BEQ    EVA       102A   27ED
19          LDAA   #$03      102C   8603
20          STAA   PORTA     102E   B78004
21          BRA    EVA       1031   20E6
```

Wie auch im Beispiel 4.2 gezeigt, ist es oft üblich, zu Beginn eines Programms die benutzen symbolischen Adressen genau zu definieren. In vielen Assemblersprachen wird hierfür die Abkürzung **EQU** verwendet. Dem als Ausgang zu verwendenden Port A wird die symbolische Adresse **PORTA** und dem Port B, das Eingänge aufnehmen soll, **PORTE** zugewiesen.

Die ersten 8 Befehle in diesem Programm dienen also der Initialisierung des Ein-/Ausgabebausteins. Durch den ersten Befehl werden alle

Bit des Kontrollregisters A gelöscht. An sich wäre es nur nötig gewesen (wie oben erwähnt), das Bit Nr. 3 im Kontrollregister zu löschen, um Zugang zum Datenrichtungs-Register zu erhalten, da die anderen Bit im Kontrollregister, die natürlich auch ihre Aufgaben haben, hier nicht berücksichtigt zu werden brauchen. Es ist aber einfacher, durch den **CLR**-Befehl alle Bit zu löschen. Mit dem zweiten und dritten Befehl werden dann alle Bit im Datenrichtungs-Register, das für das Port A zuständig ist, auf **1** gesetzt, womit das Port A als Ausgang geschaltet wurde.

Mit dem vierten und fünften Befehl wird dann im Kontrollregister A das Bit Nr. 3 wieder auf **1** gesetzt, so daß die Adresse **PORT A** jetzt als Ein- und Ausgabe angesprochen werden kann.

Durch den sechsten und siebenten Befehl wird in gleicher Weise zunächst das Kontrollregister B und dann das Datenrichtungs-Register, das für Port B zuständig ist, auf **0** gesetzt. Hierdurch wird Port B als Eingang geschaltet. Der achte Befehl setzt dann wieder das Bit Nr. 3 im Kontrollregister B durch Ausgabe des noch im Akkumulator stehenden Bitmusters **04** auf **1,** womit die Adresse **PORTE** für Ein- und Ausgänge angesprochen werden kann.

Die weiteren Befehle 9 bis 21 entsprechen im wesentlichen den Befehlen des Beispiels 4.1. Bei diesem Prozessor wird durch die Testbefehle **BITA** jeweils der Inhalt des Akkumulators A mit dem Inhalt der angesprochenen Adresse verglichen. Dieser Befehl bewirkt eine UND-Verknüpfung, wie im Bild 5.6 gezeigt.

Wenn an den zu prüfenden Bit-Positionen sowohl im Akkumulator als auch im zu prüfenden Bitmuster eine **1** steht, erscheint nach der Verknüpfung durch den Befehl im Ergebnis auch eine **1**. Das heißt, das Ergebnis ist nicht **00.**

Der Sprung zur erneuten Abfrage wird immer dann ausgeführt, wenn das Ergebnis der Prüfung **00** ist, also keine Übereinstimmung der Bitmuster bestand.

Es sei darauf hingewiesen, daß diese Art der Bit-Prüfung nur für die Prüfung eines einzelnen Bit in einem Bitmuster geeignet ist. Wollte man zwei Bit gleichzeitig auf den Wert **1** prüfen, so geht das nicht, weil das Ergebnis auch dann ungleich **00** ist, wenn nur eins der beiden zu prüfenden Bit den Wert **1** hat.

Aus diesem Grund wird im zweiten Teil des Programms auch nur das Bit Nr. 7 geprüft und nicht wie im Programmbeispiel 4.1 gleichzeitig die

Bit Nr. 6 und 7. Die Aufgabe wird aber trotzdem richtig erfüllt, da das Programm in einer Schleife im ersten Teil auch immer das Bit Nr. 6 mitprüft.

6502 Im Beispiel 4.3, das für den *6502* geschrieben ist, wird angenommen, daß der Ein-/Ausgabebaustein *6530* Verwendung findet. Dieser Baustein ist mit 4 Adressen ansprechbar, je eine davon für das Port A, für das Port B, für das Richtungsregister zum Port A und für das Richtungsregister zum Port B.

Im Programm wird das Port A als Ausgang und das Port B als Eingang benutzt. Entsprechende symbolische Adressen sind am Programmanfang festgelegt. Auch hier bedeutet eine **1** im Datenrichtungsregister, daß die Leitung des Port, als Ausgang geschaltet sind, und eine **0**, daß sie als Eingang dient.

4.3 6502

PIO 6530 8000 . . . 8003

```
                Ausgang A       PORTA   EQU   $8000
                Eingang B       PORTE   EQU   $8002
                Richt.-Reg. A   RIRA    EQU   $8001
                Richt.-Reg. B   RIRB    EQU   $8003

 1              LDA   #$FF     1000    A9FF
 2              STA   RIRA      1002    8D0180
 3      ELLI    LDA   #$00     1006    A900
 4              STA   RIRB      1008    8D0380
 5              STA   PORTA     100B    8D0080
 6      EVA     BIT   PORTE     100E    2C0280
 7              BVC   ELLI      1011    50F3
 8              LDA   #01       1013    A901
 9              STA   PORTA     1015    8D0080
10              LDA   PORTE     1018    AD0280
11              CMP   #$30      101B    C930
12              BNE   EVA       101D    D0EF
13              LDA   #$03      101F    A903
14              STA   PORTA     1021    8D0080
15              JMP   EVA       1024    4C0E10
```

Durch die ersten 4 Befehle erfolgt die Programmierung der beiden Datenrichtungs-Register. Das eigentliche Programm beginnt dann mit dem fünften Befehl, indem die Ausgangsleitungen auf **0** gesetzt werden. Als nächstes kommt dann der Testbefehl, der beim *6502* noch eine zusätzliche Möglichkeit der Prüfung bietet, die im Bild 4.20 schon beschrieben wurde. Auch bei diesem Befehl erfolgt der Test durch UND-Verknüpfung mit dem Akkumulator, was hier aber nicht interessiert. Außerdem werden, wie das Bild 4.20 zeigt, die Bit Nr. 6 und 7 des zu prüfenden Bitmusters in das Programm-Status-Wort übernommen und können dort durch einen Sprungbefehl abgefragt werden. Der siebente Befehl **BVC** bestimmt also, daß immer dann zurückgesprungen werden soll, wenn das Überlaufbit im Programm-Status-Wort gelöscht ist. Da nun durch den vorhergehenden **BIT**-Befehl die im Bit Nr. 6 stehende Information in das Überlaufbit des Programm-Status-Wortes geladen wurde, erfolgt der Rücksprung immer dann, wenn am Eingangsport die Leitung am Bit Nr. 6 noch nicht auf **1** gesetzt, d.h. also, der Schalter S 1 noch nicht eingeschaltet wurde.

Beim zweiten Teil des Programms wird eine andere Prüfmethode verwendet. Hier wird der **CMP**-Befehl, also ein Vergleichsbefehl, benutzt, durch den eine Subtraktion mit dem Hexadezimalwert **CO** ausgeführt wird.

Wenn die Eingangsinformation auch den Wert **CO** hat, dann ist das Ergebnis der Subtraktion durch den Vergleichsbefehl **00.** Im nächsten Befehl wird bestimmt, daß gesprungen werden soll, wenn der vorherige Befehl nicht zum Ergebnis **00** geführt hat, d.h. also, wenn einer der beiden Schalter nicht eingeschaltet war.

Es sei noch erwähnt, daß Vergleichsbefehle für die Prüfung einzelner Bit in einem Bitmuster nur dann Verwendung finden können, wenn sichergestellt ist, daß die nicht zu prüfenden Bit durch entsprechende vorherige Befehle oder durch Hardwareschaltung gelöscht, also auf **0** gesetzt sind.

Der *8080,* für den das gleiche Programmbeispiel 4.4 dargestellt ist, kennt besondere Befehle, durch die jede Adresse auf der Seite Null für Ein- oder Ausgaben Verwendung finden kann. Die gewählten Adressen für Eingangs- und Ausgangsport sind im Beispiel aufgeführt. ***8080***

Auch bei diesem Prozessor kann man den Akkumulatorinhalt auf **00** setzen, indem man die Exklusiv-ODER-Verknüpfung des Akkumulators mit sich selbst programmiert. Durch den **OUT**-Befehl werden die Lampe

4.4 8080

```
Eingangsport-Adresse          0020 H = PORTE
Ausgangsport-Adresse          0021 H = PORTA

 1 ELLI   XRA    A          1000  AF
 2        OUT    PORTA      1001  D321
 3 EVA    IN     PORTE      1003  DB20
 4        ANI    40H        1005  E640
 5        CPI    40H        1007  FE40
 6        JNZ    ELLI       1009  C20010
 7        MVI    A,01       100C  3E01
 8        OUT    PORTA      100E  D321
 9        IN     PORTE      1010  DB20
10        ANI    0C0H       1012  E6C0
11        CPI    0C0H       1014  FE40
12        JNZ    EVA        1016  C20310
13        MVI    A,03       1019  3E03
14        OUT    PORTA      101B  D321
15        JMP    EVA        101D  C30310
```

und das Gerät ausgeschaltet und dann durch den **IN**-Befehl der Inhalt des Eingangsports zum Akkumulator transportiert.

Für die Abfrage wird diesmal das in Bild 5.6c beschriebene Verfahren verwendet. Durch eine Maske mit UND-Verknüpfung werden zunächst alle nicht interessierenden Bit auf **0** gesetzt. Dann wird durch den folgenden Vergleichsbefehl festgestellt, ob bei einer Subtraktion der in diesem Befehl enthaltenen Maske das Ergebnis **00** oder nicht **00** ist. Der folgende Sprung an den Anfang des Programms wird ausgeführt, wenn das Ergebnis nicht **00** war.

War das Ergebnis **00,** wird der Wert **01** an das Ausgangsport gegeben, und die zweite Abfrage erfolgt in gleicher Weise.

5.5 Programmbeispiel 5

Codewandlung durch Rechnung

Lernziele:
Aufbau eines Rechen-Algorithmus. Anwenden von Vergleichsbefehlen. Bit-Maskierung. Logisches UND. Logisches ODER. Festlegen von Sprunggrenzen.

Aufgabe:
Die in der Speicherstelle **0050** *stehende Hexadezimalzahl ist in ein ASCII-Codezeichen zu wandeln und in* **0052** *abzulegen. Danach ist das Codezeichen wieder in eine Hexadezimalzahl zurückzuverwandeln, die in* **0051** *abzuspeichern ist.*

Lösungsweg:
Der ASCII-Code wird allgemein in der Computertechnik für den Datenaustausch verwendet. In einem 7 Bit breiten Muster werden alle Zahlen und Buchstaben in Groß- und Kleinschreibung sowie andere Symbole und Steuerzeichen dargestellt. (Vollständige Darstellung des ASCII-Code im Anhang.) Das höchstwertigste Bit einer 8 Bit breiten Information, das für den Code selbst nicht gebraucht wird, kann zur Darstellung der Parität verwendet werden, oder wird ständig auf **1** oder **0** gesetzt.

Die Darstellung der Hexadezimalzahlen **0** bis **F** als ASCII-Zeichen mit dem höchstwertigsten Bit als **1** ist im Programmbeispiel 5 zu sehen. Man erkennt, daß zwischen dem zweistelligen Hexadezimalwert für die ASCII-Zeichen und einer einstelligen Hexadezimalzahl ein bestimmter Zusammenhang besteht. Dieser Zusammenhang kann ausgenutzt werden, um einen *Rechen-Algorithmus,* d.h. ein gültiges Rechenverfahren, für die Umwandlung aufzustellen. Im Flußschaubild ist dieses dargestellt.

Liegt der Hexadezimalwert zwischen **0** und **9,** so ist lediglich **0B** hinzuzufügen, um zum ASCII-Zeichen zu kommen.

Ist der Hexadezimalwert größer als **9,** muß der Wert **9** subtrahiert werden, und im höherwertigen Halb-Byte ist **C** hinzuzufügen.

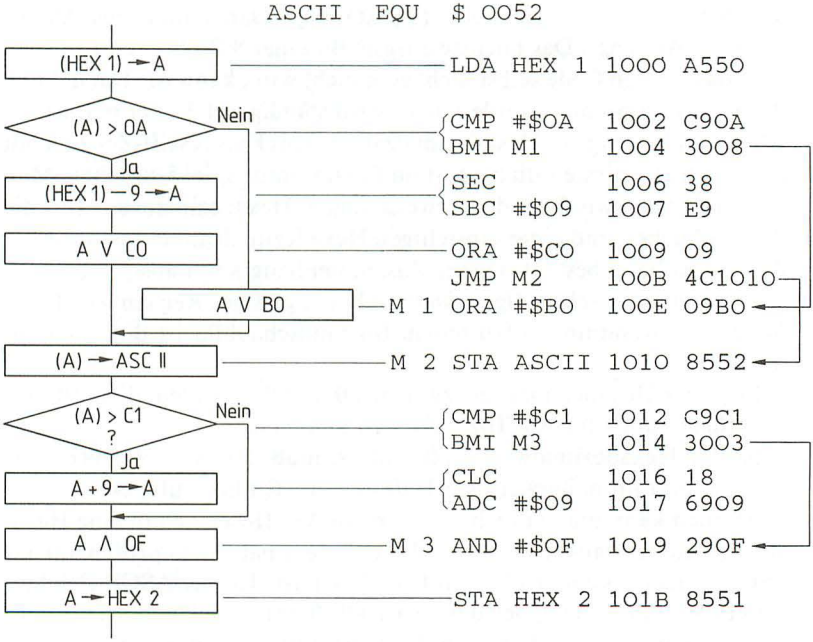
Ähnlich kann man vorgehen, wenn ein ASCII-Zeichen in eine Hexadezimalzahl gewandelt werden soll. In diesem Fall ist zu prüfen, ob das ASCII-Zeichen kleiner oder größer als **C1** ist. Ist das ASCII-Zeichen kleiner, so braucht lediglich das erste Halb-Byte ausgeblendet zu werden, ist es größer, so ist vorher noch der Wert **9** hinzuzuaddieren.

ASCII-Zeichen		Hex-zahl	ASCII-Zeichen		Hex-zahl
10110000	B0	0	10111000	B8	8
10110001	B1	1	10111001	B9	9
10110010	B2	2	11001010	C1	A
10110011	B3	3	11001011	C2	B
10110100	B4	4	11001100	C3	C
10110101	B5	5	11001101	C4	D
10110110	B6	6	11001110	C5	E
10110111	B7	7	11001111	C6	F

5.1

6502

```
HEX 1   EQU  $ 0050
HEX 2   EQU  $ 0051
ASCII   EQU  $ 0052
```



Das Beispiel 5.1 zeigt das Programm für den *6502*. Hierbei ist es notwendig, zunächst die Sprunggrenzen genau zu untersuchen und die richtige Maske und den richtigen Sprungbefehl zu wählen. Am besten stellt man hierfür eine kleine Tabelle auf und kontrolliert hiermit, ob der gewählte Befehl auch die gewünschte Wirkung veranlaßt. **6502**

(A)	08	09	0A	0B	
Maske	0A	0A	0A	0A	BMI
Ergebnis	Neg.	Neg.	Null	Pos.	
(A)	B8	B9	C1	C1	
Maske	C1	C1	C1	C1	BMI
Ergebnis	Neg.	Neg.	Null	Pos.	
	Springen	←	→	Nicht springen	

Der Sprungbefehl **BMI** wird ausgeführt, wenn das Ergebnis der Vergleichsoperation, die immer eine Subtraktion ist, negativ wird. Dementsprechend muß für den ersten **BMI**-Befehl als Maske **0A** und für den zweiten **C1** gewählt werden.

Um Teile eines Bitmusters zu einem anderen Bitmuster hinzuzufügen, benutzt man gern die ODER-Verknüpfung, die in diesem Programm auch zweimal Verwendung findet. Die UND-Verknüpfung dagegen wird im vorletzten Befehl benutzt, um das nicht interessierende obere Halb-Byte auf **0** zu stellen.

Zu beachten sind noch die Sprungbefehle. Der *6502* kennt für bedingte Sprungbefehle nur relative Adressierung; der unbedingte Sprungbefehl zur symbolischen Adresse **M2** muß dagegen mit absoluter Adressierung programmiert werden. Im ersten Teil des Programms handelt es sich um ein typisches Strukturelement für bedingten Sprung, bei dem sowohl bei der JA- als auch bei der NEIN-Entscheidung ein anderer Programmteil durchlaufen wird. In diesem Fall kann die Zusammenführung der beiden Programmteile, wie es auch in Bild 5.1 dargestellt ist, nur durch einen unbedingten Sprung erfolgen.

Das im Beispiel 5.2 gezeigte Programm für den *2650* ist im Prinzip gleich aufgebaut. Auch hier steht vor den bedingten Sprungbefehlen ein Vergleichsbefehl, dessen Ergebnis den Bedingungscode im unteren Pro- **2650**

5.2 2650

```
                HEX 1    EQU   H'0050'
                HEX 2    EQU   H'0051'
                ASCII    EQU   H'0052'

 1      CPSL     H'08'          1000   7508
 2      LODA,0   HEX 1          1002   0C0050
 3      COMI,0   H'09'          1005   E409
 4      BCFR,1   M1             1007   9906
 5      SUBI,0   H'09'          1009   A409
 6      IORI,0   H'C0'          100B   64C0
 7      BCTR,3   M2             100D   1B02
 8 M1   IORI,0   H'B0'          100F   64B0
 9 M2   STRA,0   ASCII          1011   CC0052
10      COMI,0   H'C1'          1014   E4C1
11      BCTR,2   M3             1016   1A02
12      ADDI,0   H'09'          1018   8409
13 M3   ANDI,0   H'0F'          101A   440F
14      STRA,0   HEX 2          101C   CC0051
```

gramm-Status-Wort verändert. Für die erste Entscheidung wird der Befehl **BCFR,1** gewählt, bei dem der Sprung erfolgt, wenn die Bedingungscodes im Programm-Status-Wort und im Befehl nicht übereinstimmen. Demgegenüber erfolgt im zweiten Teil des Programms der Sprung, wenn die Bedingungscodes übereinstimmen, weil hier der Befehl **BCTR,2** gewählt wurde. Die folgende Tabelle zeigt, wann die Sprünge ausgeführt werden und wann nicht gesprungen wird:

Reg. 0	08	09	0A	0B	Springen, wenn
Maske	09	09	09	09	Codes nicht
CC	10	00	01	01	übereinstimmen
CC im Befehl	01	01	01	01	BCFR,1
	Springen ←		→ Nicht springen		

Reg. 0	B8	B9	C1	C2	Springen, wenn Codes übereinstimmen BCTR,2
Maske	C1	C1	C1	C1	
CC	10	10	00	01	
CC im Befehl	10	10	10	10	

Springen ← → Nicht springen

Hierbei muß man beachten, daß **01** im Bedingungscode bedeutet, daß der Registerinhalt größer als die Maske ist, und **10,** daß er kleiner ist.

Auch das Programm für den *6800,* das im Beispiel 5.3 dargestellt ist, sieht wieder sehr ähnlich aus. Es entspricht weitgehend dem Programm für den *6502,* da beide Prozessoren den in der Wirkung gleichen relativen Sprungbefehl **BMI** kennen, bei dem der Sprung ausgeführt wird, wenn das Vergleichsergebnis eine negative Zahl gebracht hat. Da dieser Prozessor auch einen unbedingten Sprung mit relativer Adressierung kennt, wird dieser beim sechsten Befehl verwendet.

6800

5.3 6800

```
                HEX 1   EQU   $0050
                HEX 2   EQU   $0051
                ASCII   EQU   $0052

     1          LDAA    HEX 1        1000 9650
     2          CMPA    #$0A         1002 810A
     3          BMI     M1           1004 2B06
     4          SUBA    #$09         1006 8009
     5          ORAA    #$C0         1008 8AC0
     6          BRA     M2           100A 2002
M1   7          ORAA    #$B0         100C 8AB0
M2   8          STAA    ASCII        100E 9752
     9          CMPA    #$C1         1010 81C1
    10          BMI     M3           1012 2B02
    11          ADDA    #$09         1014 8B09
M3  12          ANDA    #$0F         1016 840F
    13          STAA    HEX 2        1018 9751
```

8080 Das Beispiel 5,4, das für den *8080* geschrieben ist, zeigt wieder die Besonderheit dieses Prozessors mit der Adressierung durch das Registerpaar HL. Bei der Durchsicht dieses Programms muß man also immer im Auge behalten, welchen Inhalt dieses Registerpaar hat, denn er gilt jedesmal als Zeiger für die Operandenadresse, wenn im Operandenfeld der Buchstabe **M** steht.

5.4				8080	
1		LXI	H,HEX 1	1000	21500
2		MOV	A,M	1003	7E
3		CPI	0AH	1004	FE0A
4		JM	M1	1006	FA1010
5		SUI	09H	1009	D609
6		ORI	0C0H	100B	F6C0
7		JMP	M2	100D	C31210
8	M1	ORI	0B0H	1010	F6B0
9	M2	INX	H	1012	23
10		INX	H	1013	23
11		MOV	M,A	1014	77
12		CPI	0C1H	1015	FEC1
13		JM	M3	1017	FA1C10
14		ADI	09	101A	C609
15	M3	ANI	0FH	101C	E60F
16		DCX	H	101E	2B
17		MOV	M,A	101E	77

Etwas verwirrend ist auch bei diesem Prozessor die Schreibweise der Hexadezimalzahlen in seiner Assemblersprache. Das Kennzeichen für hexadezimal ist hier ein der Zahl nachfolgendes **H.** Außerdem besteht die Regel, daß eine **0** der Hexadezimalzahl vorangestellt werden muß, wenn diese mit einem Buchstabensymbol beginnt.

Alle Sprünge müssen bei diesem Prozessor mit absoluter Adressierung programmiert werden. Im übrigen entspricht das Programm den schon vorher beschriebenen Programmen. Zur Übung wird empfohlen, die Registerinhalte Schritt für Schritt zu verfolgen, wie es im Beispiel 1.4 gezeigt wurde.

5.6 Programmbeispiel 6

Speicherbereich auf Null setzen

Lernziele:

Verschiedene Möglichkeiten der Index-Adressierung, Berechnung einer Anzahl von Speicherplätzen, Berechnung von effektiven Adressen bei Indexbefehlen.

Aufgabe:

Der RAM-Speicherbereich von Adresse **0040** *bis* **0060** *soll auf* **00** *gesetzt werden. Danach ist zu prüfen, ob alle Speicherstellen den Wert* **00** *haben. Zur Kontrolle soll als Merker in* **006A** *das Bitmuster* **00** *gesetzt werden, wenn dies der Fall ist.*

Lösungsweg:

Bei dieser Aufgabe muß der Vorgang des Nullsetzens mehrfach wiederholt werden, so daß er zweckmäßigerweise als Schleife programmiert wird. Die gewünschte Kontrolle kann dadurch erfolgen, daß der Inhalt der auf **00** gesetzten Speicherstellen laufend addiert wird. Wenn das Nullsetzen richtig gearbeitet hat, muß als Ergebnis der Addition auch **00** herauskommen. Dieser Wert kann dann in dem Merker abgespeichert werden.

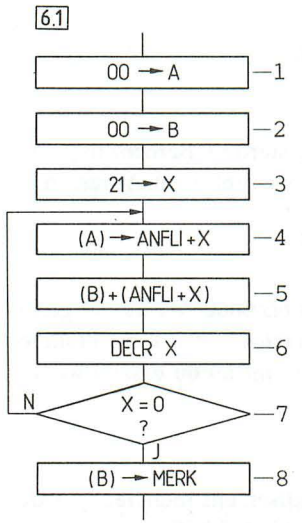
Ein Flußdiagramm für diese Programmieraufgabe ist im Beispiel 6.1 zu sehen, das die Programmierung für den *6800* zeigt. **6800**

Die erste und die letzte Speicherstelle, die auf **0** gesetzt werden sollen, sind **0040** und **0060.** Zweckmäßigerweise ist nun zunächst festzustellen, um wieviel Speicherplätze es sich hierbei handelt, was nach folgendem Schema geschehen kann:

$$
\begin{aligned}
40 \cdots 4F &= 16 \text{ Speicherplätze} = \text{Hex } 10 \\
50 \cdots 5F &= 16 \text{ Speicherplätze} = \text{Hex } 10 \\
60 &= 1 \text{ Speicherplatz} = \text{Hex } 1 \\
\hline
&33 \text{ Speicherplätze} = \text{Hex } 21
\end{aligned}
$$

Insgesamt handelt es sich also um **33** Speicherstellen, was der Hexadezimalzahl **21** entspricht und die Zahl der Schleifendurchgänge bestimmt.

Programmbeispiel 6 - Speicher auf Null setzen



6800

```
                CLRA              1000 4F
                CLRB              1001 5F
                LCX  #$0021       1002 CE0021
PAUL            STAA ANFLI-1,X    1005 A73F
                ADDB ANFLI-1,X    1007 EB3F
                DEX               1009 09
                BNE  PAUL         100A 26F9
                STAB MERK         100C D76A
```

Variante: Speicherbereich von 0240 bis 0260

```
 3      LDX  #$0240 1002 CE0240
 4 PAUL STAA #$00,X 1005 A700
 5      ADDB #$00,X 1007 EB00
 6      INX         1008 08
 7      CPX  #$0261 1009 8C0261
 8      BNE  PAUL   100C 26F7
10      STAB MERK   100E D76A
```

In den meisten Assemblersprachen ist es üblich, eine längere Liste von Speicherplätzen durch Angabe des ersten Speicherplatzes und durch Angabe der Anzahl der Speicherplätze zu reservieren. Dies geschieht zum Beispiel durch den Ausdruck

```
ANFLI EQU $ 0040        oder        ANFLI EQU H'0040'
      RMB $ 21          in anderer        RES H'21'
                        Schreibweise
```

der bedeutet, daß der Listenanfang mit der symbolischen Adresse **ANFLI** im Speicherplatz **0040** beginnt und daß für diesen Bereich insgesamt **21** Speicherplätze zu reservieren sind. Beide Zahlenangaben sind als Hexadezimalzahlen anzusehen. Bei allen weiteren Programmbeispielen wird diese Schreibweise für die Definition einer Anzahl von Speicherstellen Verwendung finden.

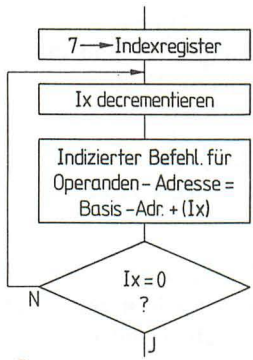
Mit den ersten Befehlen des Programms werden beide Akkumulatoren auf **00** gesetzt und danach das Indexregister mit der Anzahl der Speicherstellen, also der Zahl der Schleifendurchläufe, geladen.

Der Inhalt des Akkumulators A wird immer an die Speicherstellen ausgegeben und setzt diese damit auf **00**, während im Akkumulator B die Kontrolladdition durchgeführt wird. Danach erfolgt die Decrementierung des Indexregisters und der Rücksprung so lange, bis das Indexregister auf **00** heruntergezählt ist.

Bei den Befehlen 4 und 5 mit Index-Adressierung muß man nun allerdings sorgfältig überlegen, welche Basisadresse dort einzusetzen ist, denn der Prozessor findet bekanntlich die effektive Adresse aus der Addition der Basisadresse und dem Inhalt des Indexregisters. Anhand des Bildes 5.6 sollen die verschiedenen Möglichkeiten für den Aufbau einer Schleife mit Index-Adressierung dargestellt werden.

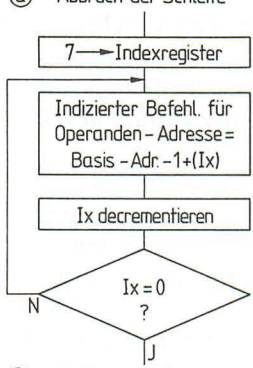
Man kann grundsätzlich eine *Liste* oder – wie man auch sagt – ein *Datenfeld* oder einen *Datenblock* durch Index-Adressierung, entweder von der niedrigsten Adresse beginnend oder von der höchsten Adresse des Blocks beginnend, abarbeiten, man kann mit Incrementierung oder Decrementierung des Indexregisters arbeiten, und man kann auch neben dem Indexregister noch einen getrennten Schleifenzähler verwenden. In Bild 5.7 sind vier verschiedene Möglichkeiten für den Aufbau einer Schleife dargestellt.

In 5.7a) und b) wird die Schleife von der höchsten Adresse beginnend abgearbeitet. Das Indexregister wird jeweils mit der Zahl der zu verar-



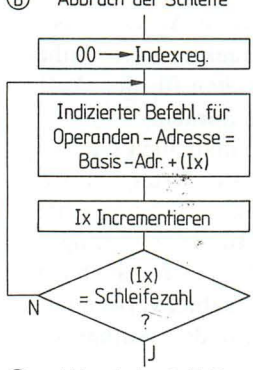
ⓐ Abbruch der Schleife

Nr. der Schleife	(Ix)	Effektive Operanden Adresse
1.	6	oo5o + 6 = oo56
2.	5	oo5o + 5 = oo55
3.	4	oo5o + 4 = oo54
4.	3	oo5o + 3 = oo53
5.	2	oo5o + 2 = oo52
6.	1	oo5o + 1 = oo51
7.	o	oo5o + o = oo5o
		Abbruch der Schleife



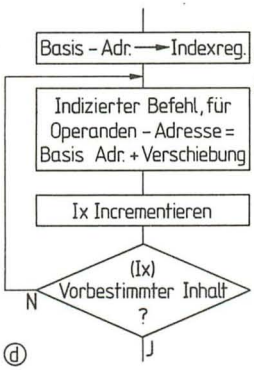
ⓑ Abbruch der Schleife

1.	7	oo4F + 7 = oo56
2.	6	oo4F + 6 = oo55
3.	5	oo4F + 5 = oo54
4.	4	oo4F + 4 = oo53
5.	3	oo4F + 3 = oo52
6.	2	oo4F + 2 = oo51
7.	1	oo4F + 1 = oo5o
	o	Abbruch der Schleife



ⓒ Abbruch der Schleife

1.	o	oo5o + o = oo5o
2.	1	oo5o + 1 = oo51
3.	2	oo5o + 2 = oo52
4.	3	oo5o + 3 = oo53
5.	4	oo5o + 4 = oo54
6.	5	oo5o + 5 = oo55
7.	6	oo5o + 6 = oo56
	7	Abbruch der Schleife



Nr. der Schleife	(Index R)	Verschiebung im Befehl	Effektive Adresse
1.	oo5o	oo	oo5o
2.	oo51	oo	oo51
3.	oo52	oo	oo52
4.	oo53	oo	oo53
5.	oo54	oo	oo54
6.	oo55	oo	oo55
7.	oo56	oo	oo56
	oo54	Abbruch der Schleife	

Bild 5.7 Berechnung der effektiven Adressen bei Befehlen mit indizierter Adressierung
a) Programmablauf einer Schleife mit Decrementierung am Beginn der Schleife
b) Programmablauf einer Schleife mit Decrementierung am Ende der Schleife
c) Programmablauf einer Schleife mit Incrementierung des Indexregisters und der Basisadresse im Befehl *(6502)*
d) Programmablauf einer Schleife mit Incrementierung des Indexregisters, Basisadresse im Indexregister *(6800)*

beitenden Speicherstellen geladen, während die Basisadresse im Befehl steht. Der Unterschied zwischen a) und b) besteht nun darin, daß einmal die Decrementierung des Indexregisters am Beginn der Schleife, d.h. vor dem Befehl mit Index-Adressierung erfolgt, während im Beispiel b) am Beginn der Schleife zunächst der Indexbefehl steht und dann erst die Decrementierung erfolgt.

Wie die Tabellen zeigen, gibt es die klarsten Verhältnisse beim Beispiel a). Hier wird die wirkliche Schleifenanzahl in das Indexregister geladen und die niedrigste Adresse als Basisadresse verwendet. Schon vor dem ersten Indexbefehl wird einmal decrementiert, so daß durch den nachfolgenden Indexbefehl richtig die letzte Adresse des Blocks angesprochen wird.

Wird mit Decrementierung am Ende der Schleife gearbeitet, muß als Basisadresse eine Adresse eingesetzt werden, die um den Wert **1** kleiner ist als die Anfangsadresse des Blocks.

Wenn der Block, beginnend von der niedrigsten Adresse aus, verarbeitet werden soll, wie es in c) dargestellt ist, muß das Indexregister zunächst

mit **00** geladen werden. Bei jedem Schleifendurchlauf wird es dann incrementiert. Der Abbruch der Schleife erfolgt, wenn der Inhalt des Indexregisters gleich der Schleifenzahl ist.

Noch eine etwas andere Möglichkeit, eine Schleife für Index-Adressierung aufzubauen, ergibt sich beim *6800,* da dessen Indexregister eine vollständige Adresse mit 16 Bit aufnehmen kann. Wie das Beispiel d) zeigt, wird hier die vollständige Basisadresse in das Indexregister geladen. Die zusätzlich gewünschte Verschiebung kommt in den Indexbefehl und beträgt im Beispiel **00.** Nach dem Befehl wird das Indexregister incrementiert, bis sein Inhalt um **1** größer ist als die Adresse des letzten Speicherplatzes, der im Block zu verarbeiten ist. Dann erfolgt der Schleifenabbruch.

Zurück zum Programmbeispiel 6.1. Hier ist die Schleife entsprechend dem Beispiel b) aufgebaut, so daß als Basisadresse in die Befehle 4 und 5 die Adresse unmittelbar unterhalb der Adresse des Listenanfangs einzusetzen ist. Das **X** hinter dem Komma kennzeichnet die beiden Befehle als Befehle mit Index-Adressierung.

Diese Art des Schleifenaufbaus ist beim *6800* allerdings nur möglich, wenn die durch Indexbefehle anzusprechenden Adressen im Bereich der Seite Null liegen. Alle Befehle mit Index-Adressierung haben nur eine Länge von 2 Byte, so daß sie nicht eine volle Basisadresse aufnehmen können.

Liegt der zu verarbeitende Datenblock in einem Speicherraum außerhalb der Seite Null, verwendet man beim *6800* eine Schleife gemäß Beispiel d), wie dies auch als Variante im Beispiel 6.1 dargestellt ist. Die Basisadresse wird hier in das Indexregister geladen, die Befehle mit Index-Adressierung erhalten als Verschiebungswert 00. Das Indexregister wird laufend incrementiert, und der Abbruch der Schleife erfolgt, wenn das Indexregister um den Wert **1** größer ist als der letzte zu verarbeitende Speicherplatz.

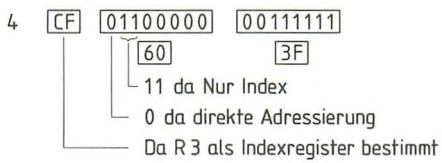
2650 Das Programm für den *2650,* das im Beispiel 6.2 dargestellt ist, verwendet auch den Schleifenaufbau entsprechend Bild 5.7b). Es entspricht im wesentlichen dem vorherigen Beispiel. Das Register 3 wird als Indexregister verwendet. Die Decrementierung des Registers 3 und der bedingte Rücksprung, wenn dieses noch nicht den Wert **00** erreicht hat, erfolgt durch den Befehl **BDDR,3.**

Der *2650* fordert eine besondere Codierung der Indexbefehle, bei der gewisse Regeln genau eingehalten werden müssen. Zunächst wird bei

```
6.2                 2650
 ANFLI  ORG H'0040' Listenanfang
        RES H'21'
 MERK   EQU H'006A' Merker


1       LODI,1 H'00'       1000 0500
2       LODI,3 H'21'       1002 0721
3  PAUL EORZ,0             1004 20
4       STRA,0 ANFLI-1,3   1005 CF603F
5       LODA,0 ANFLI-1,3   1008 0F603F
6       ADDZ,1             100B 81
7       STRZ,1             100C C1
8       BDDR,3 PAUL        100D FB76
9       STRA,1 MERK        100E CD006A
```

Codierung der Indexbefehle



allen Indexbefehlen grundsätzlich das Register 0 als Akkumulator verwendet, das heißt, dieses Register dient als Datenziel oder Datenquelle bzw. nimmt einen Operanden und später das Ergebnis bei Datenverarbeitungs-Befehlen auf. Im Assemblerausdruck der Befehle 4 und 5 steht darum nach dem Komma auch das Register 0.

Der Code für das erste Befehls-Byte richtet sich jedoch nicht nach dem Register 0, sondern nach dem für den Befehl gewählten Indexregister, das im Operandenfeld nach einem Komma steht.

Alle Indexbefehle arbeiten mit absoluter Adressierung, wobei das zweite Befehls-Byte nur den Teil der Adresse enthält, der sich auf die gerade angesprochene Adressenseite bezieht (Bild 4.3). Für die Adresse werden also nur die 5 niederstwertigen Bit des zweiten Befehls-Byte verwendet. Die Bit Nr. 5 und 6 nehmen die Indexkontrolle auf, und das Bit Nr. 7 dient als Indirekt-Bit. Das dritte Befehls-Byte enthält dann in üblicher Weise den unteren Adreßteil.

Da bei Indexbefehlen nur das Register 0 als Datenquelle-Register und als Datenziel-Register verwendet werden kann, wird zunächst der nach

dem dritten Befehl dort stehende Wert **00** in die entsprechende Speicherstelle ausgegeben und dann gleich wieder in das Register 0 geladen. Durch den sechsten Befehl werden die Inhalte der Register 0 und 1 addiert. Das Ergebnis steht im Register 0 und wird durch den siebenten Befehl im Register 1 abgespeichert. Wenn der Nullstellvorgang richtig abgelaufen ist, muß also der Inhalt des Registers 1 auch immer **00** sein.

Das Beispiel 6.3, geschrieben für den *6502*, zeigt einen etwas anderen Programmaufbau. Es werden hier zwei getrennte Schleifen, einmal für das Nullsetzen und zum zweiten für die Überprüfung durch Addition, verwendet. Die Schleifen sind gemäß Bild 5.7a) aufgebaut, und zwar mit dem Decrementbefehl zu Beginn der Schleife.

6.3 6502

```
                ANFLI   EQU   $0040
                        RES   $21

 1        LDX #$21       1000  A221
 2        LDA #00        1002  A900
 3 PAUL   DEX            1004  CA
 4        STA ANFLI,X    1005  9540
 5        BNE PAUL       1007  D0FB
 6        LDX #$21       1009  A221
 7        CLC            100B  18
 8 EGON   DEX            100C  CA
 9        ADC ANFLI,X    100D  7540
10        BNE EGON       100F  D0FB
11        STA $6A        1011  856A
```

Bei der Additionsschleife wird am Anfang durch den siebenten Befehl das Carry-Bit gelöscht. Dies ist notwendig, da beim 6502 alle Additions- und Subtraktionsbefehle das Carry-Bit mit berücksichtigen. Bei diesem Programm werden nur Befehle mit Seite-Null-Adressierung verwendet, da alle Speicherstellen in diesem Bereich liegen. Wenn der zu verarbeitende Speicherblock an anderer Stelle des Speicherraums angeordnet wäre, könnte das gleiche Programm auch Verwendung finden, nur müßte

dann der vierte und neunte Befehl auf absolute Adressierung umgestellt werden.

Der *8080,* für den das Programm im Beispiel 6.4 dargestellt ist, kennt keine Index-Adressierung im sonst üblichen Sinn. Im Beispiel wird wieder die Adressierung über das Registerpaar HL verwendet. Die Schleife wurde diesmal mit einem besonderen Schleifenzähler aufgebaut. Die Liste wird, beginnend mit der niedrigsten Adresse, abgearbeitet. Die Incrementierung des Adressenzeigers erfolgt am Ende der Schleife, und die Schleife wird verlassen, wenn das als Zähler laufende Register B bis zum Wert **00** heruntergezählt wurde, das heißt, wenn 21 Schleifenläufe erfolgt sind. **8080**

6.4 8080

```
                        ANFLI  EQU   0040H
     Listenanfang              RES     21H
     Merker             MERK   EQU   006AH

        MVI     B,21        1000  0621
        XRA     A           1002  AF
        LXI     H,ANFLI     1003  214000
PAUL    MOV     M,A         1006  77
        ADD     M           1007  86
        INX     H           1008  23
        DCR     B           1009  05
        JNZ     PAUL        100A  C20610
        STA     MERK        100D  326A00
```

Zur Vereinfachung wird in diesem Programm der Akkumulator sowohl für die Ausgabe des Wertes **00** als auch für die Ausgabe der Additionsergebnisse verwendet. Wenn also an irgendeiner Stelle beim Nullsetzen ein Fehler vorkommt, werden auch alle nachfolgenden Speicherstellen nicht auf Null gesetzt. Da die Aufgabenstellung aber nur eine Meldung und keine Lokalisierung eines Fehlers vorsah, spielt dies keine Rolle.

5.7 Programmbeispiel 7

Datenfeld (Block) im Speicher durchsuchen

Lernziele:

Definition eines Datenfeldes. Einrichtung einer Suchschleife. Aufbau eines Adreß-Zeiger-Registers.

Aufgabe:

Ein Block, mit der Adresse **2FE0** *beginnend, der* **80** *Speicherplätze einnimmt, ist darauf zu prüfen, ob darin auch ein im Akkumulator stehendes Zeichen vorkommt. Wenn ja, ist die Adresse dieses Zeichens in die Speicherstellen* **0080** *(unterer Teil) und* **0081** *auszugeben. Wenn das Zeichen nicht gefunden wird, ist dort* **00** *zu laden.*

Lösungsweg:

Auch bei diesem Programm wird zweckmäßigerweise mit indizierter Adressierung gearbeitet. Es wird der Inhalt einer Speicherstelle nach der anderen mit dem Akkumulator verglichen. Wenn beides übereinstimmt, wird die dann gerade gültige Adresse, nämlich der Wert, der sich aus der Adresse des Listenanfangs plus dem Inhalt des augenblicklichen Standes des Indexregisters ergibt, in die Speicherstelle **ZEIAD** geladen.

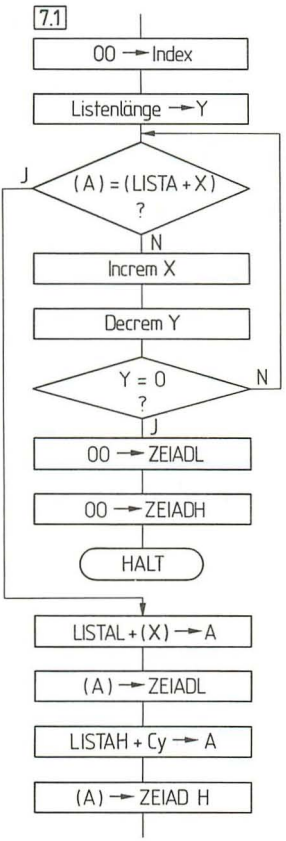
6502 Im Beispiel 7.1, das für den *6502* geschrieben ist, werden in der Schleife beide Indexregister verwendet. Das Register X dient hier als eigentliches Indexregister und wird im Anfang auch mit **00** geladen, das Register Y dient als Schleifenzähler. Es wird mit dem Hexadezimalwert **50,** der dem Dezimalwert **80,** also der Zahl der zu durchsuchenden Speicherplätze entspricht, geladen.

Die Zählschleife ist gemäß Bild 5.7b) mit Decrementierung am Ende aufgebaut. Sie muß also mit der vollen Zahl der Speicherplätze geladen werden. Die gleichzeitig mit verarbeitete Indexschleife ist aber gemäß c) angeordnet. Es wird also zunächst der niedrigste Speicherplatz verarbeitet und seine Adresse als Basisadresse in den Befehl eingesetzt.

Wenn der Schleifenzähler den Wert **00** erreicht hat, ohne daß vorher ein Aussprung aus der Schleife erfolgt ist, werden die Adressen **0080** und **0081** mit **00** geladen.

Der Aussprung aus der Schleife erfolgt durch den fünften Befehl, wenn vorher im Vergleichsbefehl Übereinstimmung des im Speicher

Programmbeispiel 7 - Blockdurchsuchen



6502

```
LISTA   EQU $2FEO Listenanfang
        RES    80 = Hex 50
ZEIAD   EQU $0080 Zeichenadr.Low
        RES     2
```

1		LDX #00	1000	A200
2		LDY #$50	1002	A050
3		LDA #$ZEICHEN	1004	A9XX
4	EVA	CMP LISTA,X	1006	DDE02F
5		BEQ GUT	1009	F00B
6		INX	100B	E8
7		DEY	100C	88
8		BNE EVA	100D	D0F7
9		LDA H00	100F	A900
10		STA ZEIAD	1011	8580
11		STA ZEIAD+1	1013	8581
12		BRK	1015	00
13	GUT	TXA	1016	8A
14		CLD	1017	D8
15		CLC	1018	18
16		ADC #$E0	1019	69E0
17		STA ZEIAD	101B	8580
18		LDA #$2F	101D	A92F
19		ADC #00	101F	6900
20		STA ZEIAD+1	1021	8581

stehenden Bitmusters mit dem Bitmuster im Akkumulator festgestellt wurde. Zunächst wird bei der symbolischen Adresse **GUT** durch den **TXA**-Befehl der Inhalt des Indexregisters in den Akkumulator transportiert. Die Befehle 14 und 15 dienen nur dazu, die nachfolgende Addition vorzubereiten, indem der Prozessor auf binäre Additionen programmiert und das Carry-Bit gelöscht wird.

Im 16. Befehl wird der untere Adreßteil der Anfangsadresse des zu verarbeitenden Datenblocks zum Inhalt des Akkumulators hinzuaddiert. Das Ergebnis entspricht dem unteren Adreßteil des Speicherplatzes, in dem das gesuchte Zeichen gefunden wurde. Es wird also nach **ZEIAD** abgespeichert.

7.2 2650

```
    LISTA   EQU   H'1FE0'   Listenanfang
            RES     H'50'   Listenlänge
    ZEIAD   EQU   H'0080'   Zeichenadr. H
            RES        2

 1          PPSL    H'08'        1000 7708
 2          CPSL    H'0I'        1002 7501
 3          LODI,3  H'50'        1004 0750
 4          LODI,2  ZEICHEN      1006 06XX
 5 EVA      LODA,0  LISTA,3-     1008 0F5FE0
 6          COMZ,2               100B E2
 7          BCTR,0  GUT          100C 180A
 8          BRNR,3  EVA          100E 5B78
 9          EORZ,0               1010 20
10          STRA,0  ZEIAD        1011 CC0080
11          STRA,0  ZEIAD+1      1014 CC0081
12          HALT                 1017 40
13 GUT      LODI,0  H'E0'        1018 04E0
14          ADD2,3               101A 83
15          STRA,0  ZEIAD        101B CC0080
16          LODI,0  H'1F'        101E 041F
17          ADDI,0  00           1020 8400
18          STRA,0  ZEIAD+1      1022 CC0081
```

In den drei folgenden Befehlen wird das gleiche für den oberen Adreßteil wiederholt, wobei automatisch das Carry-Bit mit berücksichtigt wird, wenn bei der vorherigen Addition ein Übertrag entstanden war.

2650

Das gleiche Programm, für den *2650* geschrieben, arbeitet ohne Schleifenzähler, wie das Beispiel 7.2 zeigt. Durch die ersten beiden Befehle wird der Prozessor zunächst initialisiert, er muß mit Carry arbeiten, und das Carry-Bit muß auf **0** gesetzt sein. Als Indexregister wird das Register 3 verwendet, in das der Hexadezimalwert **50** geladen wird. Das zu suchende Zeichen kommt in das Register 2.

Der fünfte Befehl ist ein Datentransport-Befehl mit Index-Adressierung, bei dem der Inhalt des Indexregisters 3 vor Befehlsausführung um den Wert **1** vermindert wird, was durch das Minuszeichen am Schluß des Operandenfeldes ausgedrückt wird. Es handelt sich also um eine Indexschleife mit Decrementierung am Schleifenanfang. Der Datenblock wird von der letzten Adresse beginnend verarbeitet.

Die übrigen Befehle entsprechen weitgehend den Befehlen, die beim Programm des *6502* besprochen wurden. Da der *2650* auch Additionen, bei denen beide Summanden in Registern stehen, zuläßt, wird hier das Register 0 mit dem unteren Adreßteil der Anfangsadresse geladen und dazu der Inhalt des Registers 3 hinzuaddiert.

6800

Beim Beispiel 7.3, geschrieben für den *6800*, ist zu berücksichtigen, daß dieser Prozessor ein 16-Bit-Indexregister hat. Die Schleife ist also ähnlich wie im Bild 5.6c) dargestellt aufgebaut, jedoch wird zusätzlich noch der Akkumulator B als Schleifenzähler verwendet, wie es schon im Beispiel 7.1 beschrieben wurde.

Der zweite Teil des Programms, durch den die Adresse des gefundenen Zeichens abgespeichert werden soll, wird hier sehr kurz und besteht nur aus dem 13. Befehl, durch den der Inhalt des Indexregisters in die absolut adressierte Adresse und in die dann folgende Adresse abgespeichert wird. Allerdings kommt, wie es bei diesem Prozessor üblich ist, der obere Adreßteil in die höherwertige Speicheradresse, so daß sich eine geringfügige Abweichung von der Aufgabenstellung ergibt.

7.3 6800

```
                LISTA   EQU   $2FE0
                        RBM   $  50
                ZEIAD   EQU   $0080
                        RBM      2

 1              LDX     #$2FE0      1000 CE2FE0
 2              LDAA    #ZEICHEN    1003 86XX
 3              LDAB    #$50        1005 C650
 4 EVA          CMPA    00,X        1007 A100
 5              BEQ     GUT         1009 270C
 6              INX                 100B 08
 7              DECB                100C 5A
 8              BNE     EVA         100D 26F8
 9              CLRA                100F 4F
10              STAA    ZEIAD       1010 B70080
11              STAA    ZEIAD+1     1013 B70081
12              WAI                 1016 3E
13 GUT          STX     ZEIAD       1017 FF0080
```

8080 Beim gleichen Programm für den *8080,* dargestellt im Beispiel 7.4, wird wieder das Registerpaar HL als Adressenzeiger verwendet. Das Register C dient als Schleifenzähler. Der Datenblock wird mit dem ersten Zeichen beginnend verarbeitet.

Da das Registerpaar HL die komplette Adresse des gefundenen Zeichens enthält, wird auch nur ein Befehl gebraucht, um diese Adresse in zwei aufeinanderfolgende Speicherplätze abzuspeichern. Wie bei diesem Prozessor üblich, kommt der untere Adreßteil, der im Register L steht, in die im Befehl genannte Adresse, und der Inhalt des Registers H, der den oberen Adreßteil darstellt, in die nächstfolgende, so daß die Reihenfolge, die in der Aufgabenstellung genannt wurde, eingehalten wird.

7.4 8080

```
                LISTA   EQU   2FE0H
                        RES     50H
                ZEIAD   EQU   0080H
                        RES      2

 1        LXI       H,LISTA     1000 21E02F
 2        MVI       A,ZEICHEN   1003 3EXX
 3        MVI       C,50H       1005 0E50
 4 EVA    CMP       M           1007 BE
 5        JZ        GUT         1008 CA1710
 6        INX       H           100B 23
 7        DCR       C           100C 0D
 8        JNZ       EVA         100D C20710
 9        XRA       A           1010 AF
10        STA       ZEIAD       1011 328000
11        STA       ZEIAD+1     1014 328100
12        HLT                   1017 76
13 GUT    SHLD      ZEIAD       1018 228000
```

5.8 Programmbeispiel 8

Bedingter Einsprung in Unterprogramme

Lernziele:

Anwenden von Rotationsbefehlen. Indirekte Adressierung. Tabelle mit Einsprung-Adreßzeigern. Einsprung- und Rücksprungbefehle für Unterprogramme.

Aufgabe:

Je nachdem, welches der 8 Bit in einem Eingangsport gesetzt ist, soll jeweils ein anderes Unterprogramm angesprungen werden.

Lösungsweg:

Für die Lösung dieser Aufgabe gibt es mehrere Möglichkeiten, von denen einige in den Programmen für die verschiedenen Prozessoren vorgestellt werden. Je nach Lösungsweg und nach verwendetem Prozessor wird man

die *Tabelle,* in der die *Einsprungadressen* für die 8 verschiedenen Unterprogramme festgelegt sind, unterschiedlich aufbauen. Man muß dabei berücksichtigen, daß der Prozessor am Ende des Unterprogramms durch den Rücksprungbefehl wieder richtig in das Hauptprogramm zurückgeführt wird.

Um die Lernprogramme möglichst übersichtlich zu halten, wurde eine eventuell notwendige Initialisierung der Ein-/Ausgangsbausteine, die ja bereits beim Programmbeispiel 4 ausführlich beschrieben wurde, hier nicht noch einmal wiederholt.

8080 Die für den *8080* verwendete Adressentabelle ist im Beispiel 8.1 gezeigt. Sie beginnt bei **0100** und endet bei **010F.** Für jede Einsprungadresse werden 2 Speicher-Byte benötigt, wobei der untere Adreßteil der Einsprungadresse jeweils vor dem oberen Teil gespeichert wird.

Wie das Flußdiagramm zeigt, erfolgt die Ermittlung der richtigen Adresse mit Hilfe eines Rotationsbefehls. Da angenommen wird, daß jeweils nur ein Unterprogramm angewählt werden kann, das heißt, im Eingangsport darf nur ein Bit auf **1** gesetzt sein, wird das aus dem Eingangsport in den Akkumulator eingelesene Bitmuster so lange nach rechts rotiert, bis das eingelesene Bit im Carry steht. So lange das Carry-Bit noch den Inhalt **0** hat, wird das HL-Register jeweils 2mal incrementiert und zeigt dann auf den unteren Teil der nächsten Einsprungadresse eines Unterprogramms. Dann erfolgt ein unbedingter Rücksprung, und es wird erneut der Inhalt des Akkumulators um eine Position nach rechts rotiert.

Wurde festgestellt, daß das Carry-Bit auf **1** gesetzt ist, so erfolgt der Aussprung aus der Schleife zur Adresse **ADAUS,** wo ein Befehl steht, mit dem der Inhalt der Speicherstelle, auf den das Registerpaar HL zeigt, in das Register E geladen wird. Es folgt ein Increment-Befehl für das Registerpaar HL, und mit dem 12. Befehl wird der obere Teil der Einsprungadresse in das Register D geladen. Damit enthält das Registerpaar DE jetzt die Einsprungadresse, die durch die beiden Austauschbefehle **XCHG** und **PCHL** zuerst in das Registerpaar HL und dann in den Befehlsadreßzähler transportiert wird. Dies bedeutet, daß als nächstes der Befehl ausgeführt wird, der an der Adresse steht, die der Einsprungadresse des angewählten Unterprogramms entspricht.

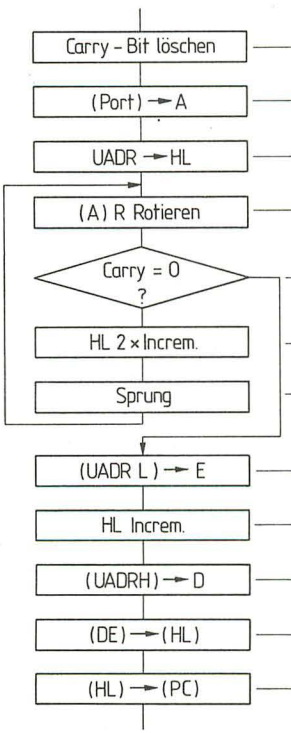
Damit der gewählte Lösungsweg richtig funktioniert, müssen die Unterprogramme in ganz bestimmter Form im Speicher angeordnet sein, wie dies im Bild 5.8 dargestellt ist.

8.1

8080

Bit Nr.	Adressen der Einsprungadressen LOW	HIGH
0	0100	0101
1	0102	0103
2	0104	0105
3	0106	0107
4	0108	0109
5	010A	010B
6	010C	010D
7	010E	010F

```
PORT   EQU 23H    Eingang
ADLIS  EQU 0100H  Adr.-Liste
       RES 16D
```




```
1          STC            1000 37
2          CMC            1001 3F
3  EIN     IN    23H      1002 DB23
4          LXI   H,0100H  1004 210001
5  SUCH    RRC            1007 0F
6          JC    AD AUS   1008 DA1010
7          INX   H        100B 23
8          INX   H        100C 23
9          JMP   SUCH     100D C30710
10 ADAUS   MOV   E,M      1010 5E
11         INX   H        1011 23
12         MOV   D,M      1012 56
13         XCHG           1013 EB
14         PCHL           1014 E9
```

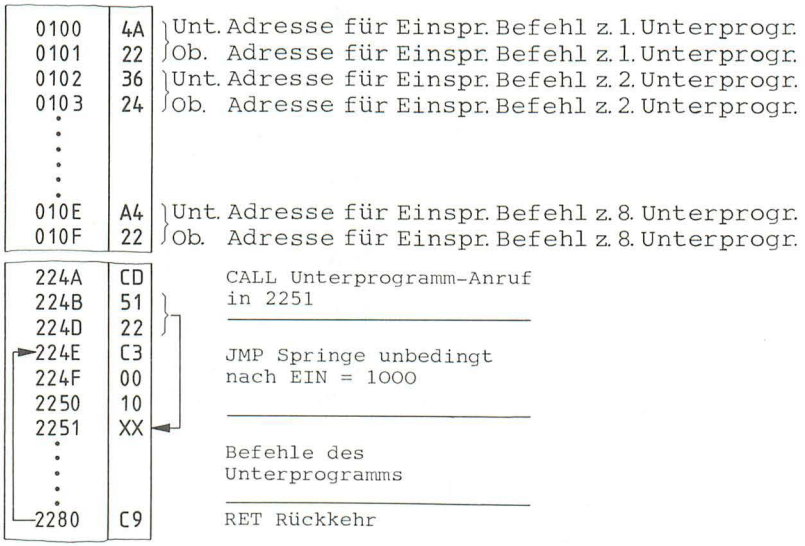


Bild 5.8 Speicherbelegung für Programmbeispiel 8.1

In der Einsprungtabelle stehen die Adressen für die Einsprungbefehle für das betreffende Unterprogramm, und zwar jeweils der untere Adreßteil zuerst. Wenn also gleich das erste Unterprogramm angewählt wurde, ist die Einsprungadresse **224A,** wo der **CALL**-Befehl für den Unterprogramm-Anruf steht. Die beiden nächsten Adreß-Bytes (**224B** und **224D**) enthalten die Adresse, in der das eigentliche Unterprogramm beginnt. Der dann in den Adressen **224E** bis **2250** folgende Befehl ist ein unbedingter Rücksprungbefehl zum Beginn des Hauptprogramms, denn es muß sichergestellt werden, daß der Prozessor nach Abarbeitung des Unterprogramms das Hauptprogramm erneut abfragt, ob ein neues Unterprogramm zu bearbeiten ist. Bekanntlich wird die dem Unterprogramm-Anruf folgende Befehlsadresse im Stapelspeicher gespeichert und von dort in den Befehlsadreßzähler geladen, wenn im Unterprogramm der **RETURN**-Befehl erscheint.

Es muß noch darauf hingewiesen werden, daß durch entsprechende Hardware-Schaltungen dafür gesorgt werden muß, daß immer nur ein Unterprogramm angewählt ist und daß auch nicht alle Anwahlschalter

auf **0** stehen. Denn sonst läuft das Programm falsch. Diese Bedingung könnte natürlich auch durch Software-Befehle kontrolliert werden.

2650

Ein etwas anderer Programmaufbau wird in dem Beispiel 8.2 für den *2650* verwendet. Dieser Prozessor hat umfangreiche Möglichkeiten für indirekte Adressierung, die in diesem Beispiel ausgenutzt werden. Als Zeiger für die indirekten Adressen werden die Speicherstellen **0080** und **0081** mit dem Symbol **ZEIG** verwendet.

Zweckmäßig ist es, bei einem Programm, das auf dem *2650* laufen soll, die Einsprungtabelle auch in umgekehrter Reihenfolge aufzubauen als beim *8080*. Hier sollte also die erste Speicherstelle immer den oberen Teil und die dann folgende Speicherstelle den unteren Teil der Adresse, in dem das betreffende Unterprogramm beginnt, enthalten.

Der erste Befehl bestimmt, daß ohne Carry gearbeitet werden soll, denn die Abfrage erfolgt hier nicht über das Carry-Bit, sondern mit dem Testbefehl **TMI.**

Das Register 1 nimmt das Bitmuster aus den Schaltern auf, und das Register 3 wird als Indexregister verwendet.

Bei diesem Programm wird durch den dritten Befehl überprüft, ob überhaupt einer der Schalter für die Eingabe betätigt wurde. Ist dies nicht der Fall, d.h., wenn im Bedingungscode **00** steht, erfolgt ein Rücksprung zur Marke **NEU.**

Im vierten Befehl wird das Indexregister mit **FE** geladen. Es steht dann nach dem fünften Befehl, in dem der Wert **02** hinzuaddiert wird, auf **00.**

Im sechsten Befehl wird das Register 1 nach rechts rotiert und anschließend der Inhalt mit der Maske **80** verglichen. Stimmt Maske und Inhalt des Registers 1 nicht überein, erfolgt der Sprung zurück zur Marke **SUCH.** Bei Übereinstimmung, d.h. wenn vor der Rotation das Bit Nr. 1 gesetzt war, wird der Inhalt der durch die Indizierung angesprochenen Speicherstelle, der den oberen Adreßteil für das anzurufende Unterprogramm enthält, zunächst in das Register 0 geladen und von dort in die Speicherstelle **ZEIG** abgespeichert. Bei den nächsten beiden Befehlen geschieht das gleiche mit dem unteren Adreßteil.

Der 13. Befehl ist ein unbedingter Einsprungbefehl in ein Unterprogramm mit indirekter Adressierung. Dies bedeutet, daß die Startadresse des Unterprogramms in der Speicherstelle und in der Stelle danach zu finden ist, die im Befehl in absoluter Adressierung genannt ist.

Nach dem Durchlaufen des Unterprogramms wird durch den Rück-

8.2 2650

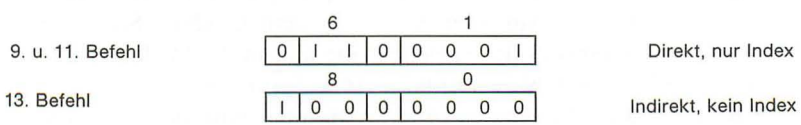
Tabelle der Unterprogramm-Startadressen

Bit 0 = 1	0100 0101	41 2C	Ob. Adr. Unt. Adr.	für Start 1. Unterprogramm
Bit 1 = 1	0102 0103	41 3D	Ob. Adr. Unt. Adr.	für Start 2. Unterprogramm
	⋮			
Bit 7 = 1	010E 010F	43 00	Ob. Adr. Unt. Adr.	für Start 8. Unterprogramm

```
                    Eingang = PORT DATA
                    UADR   EQU   H'0100'
                           RES     16
                    ZEIG   EQU   H'0080'
                           RES      2

 1          CPSL    H'80'          1000   7580
 2  NEU     REDD,1                 1002   71
 3          BCTR,0  NEU            1003   187D
 4          LODI,3  H'FE'          1005   07FE
 5  SUCH    ADDI,3  H'02'          1007   8702
 6          RRR,1                  1009   51
 7          TMI,1   H'80'          100A   F580
 8          BCFR,0  SUCH           100C   9879
 9          LODA,0  UADR,3         100E   0F6100
10          STRA,0  ZEIG           1011   CC0080
11          LODA,0  UADR+1,3       1014   0F6101
12          STRA,0  ZEIG+1         1017   CC0081
13          BSTA,3  *ZEIG          101A   3F8080
14          BCTR,3  NEU            101D   1B63
```

Aufbau der zweiten Befehlsbyte



sprungbefehl im Unterprogramm die im Stapelspeicher abgelegte Adresse in das Befehls-Adreß-Register geladen, so daß das Hauptprogramm mit dem 14. Befehl fortgesetzt wird, in dem ein unbedingter Rücksprung zum Beginn des Programms steht.

Beachten muß man bei diesem Prozessor nun wieder den Aufbau des zweiten Befehls-Byte. Bei den Befehlen 9 und 11 handelt es sich um Befehle nur mit Index-Adressierung in direkter Arbeitsweise. Das Bit Nr. 7 muß darum auf **0** und die Bit Nr. 5 und 6 auf **1** gesetzt werden, womit sich unter Berücksichtigung des oberen Teils der absoluten Adresse der Hexadezimalcode **61** ergibt.

Beim 13. Befehl ist indirekte Adressierung, aber keine Indexverwendung vorgeschrieben. Es muß also das Bit Nr. 7 auf **1** und die Bit Nr. 5 und 6 auf **0** gesetzt werden. Es ergibt sich also unter Berücksichtigung der Adresse der Hexadezimalcode **80.**

Bei dem im Beispiel 8.3 dargestellten Programm für den *6800* wird wieder etwas anders gearbeitet. Die Tabelle mit den Einsprungadressen enthält für jedes Unterprogramm 3 Speicherstellen. In der ersten steht jeweils der Operationscode für einen unbedingten Sprungbefehl mit absoluter Adressierung. Die Sprungadresse ist in den beiden nachfolgenden Speicherstellen enthalten. **6800**

Auch hier wird mit den ersten beiden Befehlen überprüft, ob überhaupt ein Schalter betätigt wurde. Dann wird das Indexregister mit dem Wert **01FD** geladen, so daß es nach drei Incrementbefehlen den Wert **0100** zeigt. Für die Abfrage wird diesmal wieder das gesetzte Carry-Bit als Kriterium verwendet, und der im achten Befehl stehende Sprung wird nur ausgeführt, wenn das Carry-Bit nicht auf **1** gesetzt ist.

Ist das Carry-Bit gesetzt, erfolgt mit dem neunten Befehl ein indexierter Einsprung in die Unterprogramm-Tabelle, und zwar zu der Adresse, auf die der Inhalt des Indexregisters zeigt. Dort steht dann der Einsprungbefehl in die eigentliche Startadresse des Unterprogramms. Nach Rückkehr vom Unterprogramm wird durch den 10. Befehl ein unbedingter Sprung zum Programmanfang vorgeschrieben.

8.3 6800

Tabelle der Unterprogramm-Startadressen

Bit 0 = 1	0100	7E	Befehl für unbedingten Sprung	
	0101	41	Ob.	Startadr. des 1. Unterprogramms
	0102	2C	Unt.	
Bit 1 = 1	0103	7E	Befehl für unbedingten Sprung	
	0104	41	Ob.	Startadr. des 2. Unterprogramms
	0105	3D	Unt.	
	⋮			
Bit 7 = 1	0115	7E	Befehl für unbedingten Sprung	
	0116	43	Ob.	Startadr. des 8. Unterprogramms
	0117	00	Unt.	

```
PORT   EQU   $8000
UADR   EQU   $0100
       RBM   $  18   (=D24)
```

Vorher PORT initialisieren

```
 1 NEU   LDAA   PORT      1000  B68000
 2       BEQ    NEU       1003  27FB
 3       LDX    #$01FD    1005  CE01FD
 4 SUCH  RORA             1008  46
 5       INX              1009  08
 6       INX              100A  08
 7       INX              100B  08
 8       BCC    SUCH      100C  24FA
 9       JSR    00,X      100E  AD00
10       JMP    NEU       1010  7E1000
```

Im Programmbeispiel 8.4 für den *6502* wird wieder ein etwas anderes Verfahren vorgestellt. Die Speicherbelegung zeigt das Bild 5.9. Außer der Liste mit den Einsprungadressen, die im ROM sein muß, wird noch ein Speicherbereich im RAM als Zeiger benötigt, in den durch das Hauptprogramm die richtige Einsprungadresse eingeschrieben wird. **6502**

Das Programm beginnt, wie schon vorher beschrieben, mit einer Port-Abfrage und einem Rücksprung in eine Warteschleife, wenn kein Schalter betätigt wurde. Dann wird das Indexregister mit **FE** geladen, damit es nach zwei Incrementierungen auf **00** steht. Für die Prüfung des Bitmusters wird wieder das Carry-Bit verwendet.

8.4 6502

```
        PORT    EQU   $8000

        UADR    EQU   $0100
                RES   $  10     (= D16)
        UPZEI   EQU   $0060
                RES     3

        Vorher PORT initialisieren

 1 NEU  LDA     PORT       1000 AD0080
 2      BEQ     NEU        1003 F0FB
 3      LDX     #$FE       1005 A2FE
 4 SUCH ROR     A          1007 6A
        INX                1008 E8
 5      INX                1009 E8
 6      BCC     SUCH       100A 90FB
 7      LDA     #$4C       100C A94C
 8      STA     UPZEI      100E 8560
 9      LDA     UADR,X     1010 BD0001
10      STA     UPZEI+1    1013 8561
11      INX                1015 E8
12      LDA     UADR,X     1016 BD0001
13      STA     UPZEI+2    1019 8562
14      JSR     UPZEI      101B 206000
15      JMP     NEU        101E 4C0010
```

Tabelle für Einsprungadressen

UADR	01 00	41	= Unt.	Adresse des 1.
UADR + 1	01 01	2C	= Ob.	Unterprogramms
UADR + 2	01 02	42	= Unt.	Adresse des 2.
UADR + 3	01 03	D4	= Ob.	Unterprogramms
⋮	⋮	⋮		
UADR + 15	01 0E	46	= Unt.	Adresse des 8.
UADR + 16	01 0F	73	= Ob.	Unterprogramms

Zeiger zur Aufnahme der Unterprogramm-Einsprungadressen

UPZEI	00 60	4C	JMP	Befehl
UPZEI+ 1	00 61	XX	Unt.	Einsprungadresse
UPZEI+ 2	00 62	01	Ob.	

Bild 5.9 Speicherbelegung für Programmbeispiel 8.4

Ist das gesetzte Bit gefunden, so wird zunächst der Wert **4C**, der dem Operationscode für einen unbedingten Sprungbefehl mit absoluter Adressierung entspricht, in die Adresse des Unterprogramm-Zeigers **UPZEI** geladen.

Der neunte Befehl ist wieder ein Befehl mit Index-Adressierung. Mit ihm wird der untere Adreßteil des angewählten Unterprogramms in den Akkumulator geladen und anschließend in das zweite Byte des Unterprogramm-Zeigers abgespeichert. Nach Incrementierung des Indexregisters erfolgt der gleiche Vorgang für den oberen Adreßteil, der in die nächstfolgende Speicherstelle des Unterprogramm-Zeigers kommt.

Als 14. Befehl kommt der Einsprungbefehl zum Unterprogramm an die Adresse des Unterprogramm-Zeigers. Durch den dort stehenden Sprungbefehl wird dann das eigentliche Unterprogramm angesprungen. Wenn dort der Rücksprungbefehl erscheint, wird das Hauptprogramm mit dem 15. Befehl fortgesetzt, der einen Sprungbefehl zum Programmanfang enthält.

5.9 Programmbeispiel 9

16-Bit-Addition mit zwei Datenfeldern

Lernziele:

Verarbeiten von Datenfeldern. Mehr-Byte-Rechnung. Berücksichtigung des Übertrags. Mehrfach-Adressierung mit Index.

Aufgabe:

Es sollen 20mal je zwei 16-Bit-Binärzahlen, deren Dezimalwert zwischen **−32 767** *und* **+32 768** *liegt, addiert werden. Die Binärzahlen stehen in den Datenfeldern* **P** *und* **Q**, *die Ergebnisse sollen in das Datenfeld* **R** *abgespeichert werden.*

Speicherbelegung: Unteres Byte in oberer Adresse

P	0100	P_1H	**Q**	0128	Q_1H	**R**	0150	R_1H
	0101	P_1L		0129	Q_1L		0151	R_1L
	⋮			⋮			⋮	
	0126	$P_{20}H$		014E	$Q_{20}H$		0176	$R_{20}H$
	0127	$P_{20}L$		014F	$Q_{20}L$		0177	$R_{20}L$

Lösungsweg:

Es ist zweckmäßig, zunächst einmal klar festzulegen, in welchen Speicherstellen die Datenfelder angelegt werden sollen und in welcher Form sie aufzubauen sind. Da auch bei diesem Programm mit Index-Adressierung gearbeitet wird, sollte man die drei Datenfelder in gleicher Weise aufbauen. Im Prinzip ist es gleich, ob in der jeweils unteren Adresse das höherwertige oder das niederwertige Byte abgespeichert wird. In den Beispielen werden die Zahlen so abgespeichert, wie man sie schreibt, das heißt, in die jeweils erste Speicherstelle kommt immer das obere Byte und in die nächstfolgende Speicherstelle das niedere Byte der zu verarbeitenden Zahl bzw. des Ergebnisses. Da bei arithmetischen Mehr-Byte-Operationen wegen der Berücksichtigung des Übertrags grundsätzlich mit dem niederstwertigsten Byte einer Zahl begonnen werden muß, hat die vorgesehene Speicheranordnung den Vorteil, daß das Programm jeweils mit der höchsten Adresse des Datenfeldes, also mit Decrementierung des Indexregisters beginnend, laufen kann.

9.1 6502

```
   PANH    EQU $0100     Dez. 40 = Hex. 28
           RES $  28
   QANH    EQU $0128
           RES $  28
   RANH    EQU $0150
           RES $  28

1          CLC                 1000  18
2          CLD                 1001  D8
3          LDX   #$28          1002  A228
4  IRMA    LDA   PANH-1,X      1004  BDFF00
5          ADC   QANH-1,X      1007  7D2701
6          STA   RANH-1,X      100A  9D4F01
7          DEX                 100D  CA
8          BNE   IRMA          100E  D0F4
9          BRK                 1010  00
```

6502

Das Programm für den *6502* ist im Beispiel 9.1 dargestellt. Zunächst müssen die Adressen der Datenfelder festgelegt werden. Da es sich um dezimal **20** Worte handelt und jedes Wort 2 Byte lang ist, werden stets **40** Speicherstellen benötigt, was hexadezimal **28** entspricht. Jedes der drei Datenfelder hat also eine Länge von **28** Speicherplätzen.

Durch die ersten beiden Befehle wird das Carry-Bit gelöscht und der Prozessor auf binäres Arbeiten programmiert. Dann kommt die Länge des Datenfeldes in das Indexregister X, und in den Befehlen 4, 5 und 6 erfolgt die Addition und das Abspeichern der Ergebnisse.

Da hier eine Schleife mit nachträglicher Decrementierung verwendet wird, müssen als Basisadressen in die Befehle jeweils die Anfangsadressen der Datenfelder **−1** eingesetzt werden.

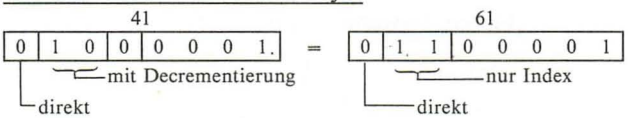
Der Übertrag wird immer bei der zweiten Operation für den gleichen Summanden mit berücksichtigt. Da die zu verarbeitenden Zahlen in ihrer Höhe begrenzt wurden, darf bei der Addition des jeweils zweiten Byte kein Übertrag entstehen. Sind die **20** Zahlen addiert, also die Schleife **40**mal durchlaufen, wird das Programm mit Befehl **BRK** angehalten.

9.2 2650

```
R3 = Indexregister  R0 = Additionsregister

1          PPSL    H'08'        1000  7708
2          CPSL    01           1002  7501
3          LODI,3  H'28'        1004  0714
4   IRMA   LODA,0  PANH,3-1     1006  0F4100
5          ADDA,0  QANH,3       1009  8F6128
6          STRA,0  RANH,3       100C  CF6150
7          BRNR    IRMA         100F  5B75
8          HALT                 1010  40
```

Aufbau der zweiten Befehlsbyte



4. Befehl 5. und 6. Befehl

Das für den *2650* geschriebene Programm, das im Beispiel 9.2 dargestellt ist, zeigt im wesentlichen den gleichen Aufbau. Der einzige Unterschied besteht darin, daß bei diesem Prozessor zusammen mit der Index-Adressierung noch eine Decrementierung des Indexregisters, und zwar vor Ausführung des Befehls, programmiert werden kann. Hiervon ist im vierten Befehl Gebrauch gemacht. Es handelt sich hier also um eine Schleife mit vorheriger Decrementierung, so daß als Basisadressen die Anfangsadressen der Datenfelder verwendet werden. **2650**

Bei der Codierung der jeweils zweiten Befehls-Byte für die Befehle mit Index-Adressierung muß man wieder die Regeln des Prozessors beachten und die Bit Nr. 5, 6 und 7 richtig einsetzen.

Im Prinzip ist auch das Programm für den *6800* in gleicher Weise aufgebaut, jedoch ist hier wieder zu berücksichtigen, daß dieser Prozessor ein 16-Bit-Indexregister hat. **6800**

Der in das Indexregister zu ladende Wert entspricht also der Anfangsadresse für das Datenfeld **P.** Die indizierten Befehle enthalten dann jeweils den Abstand der im Befehl zu verarbeitenden Adresse von der Adresse, die in das Indexregister geladen wurde.

9.3 6800

```
1          CLC              1000  0C
2          LDX   #$0100     1001  CE0100
3   IRMA   DEX              1004  09
4          LDAA  $28,X      1005  A628
5          ADCA  $50,X      1007  A950
6          STAA  $78,X      1009  A778
7          CPX   #$00D8     100B  8C00D8
8          BNE   IRMA       100E  26F4
9          WAI              1010  3E
```

Der Rücksprung kann bei diesem Prozessor nicht erfolgen, wenn das Indexregister **00** ist, da dieser Wert nicht erreicht wird. Man muß also ausrechnen, wie der Wert des Indexregisters nach den zu durchlaufenden **40** Schleifen ist. Am besten verwendet man hierfür eine kleine Hilfsrechnung:

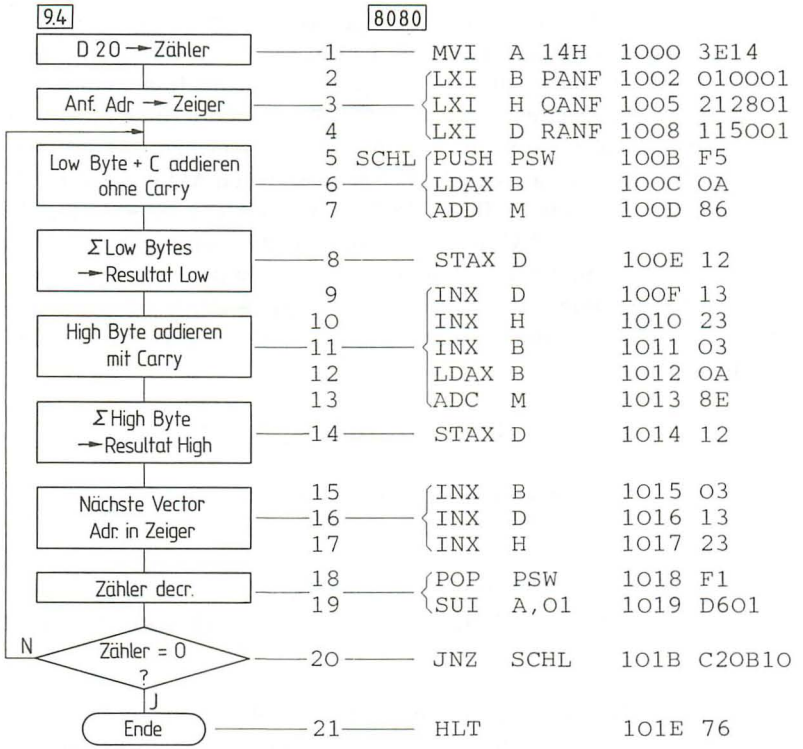
Dezimal		Hexadezimal
256	=	100
− 40	=	− 28
216	=	D8

8080 Das Programm für den *8080* ist etwas einfacher aufzubauen, wenn bei der Speicherbelegung das jeweils untere Byte der Zahl auch in die untere Adresse kommt, wie es im Beispiel 9.4 gezeigt wird.

Speicherbelegung: Unteres Byte in unterer Adresse

P_N 0100	P_1 L	Q_N 0128	Q_1 L	R_N 0150	R_1 L
0101	P_1 H	0129	Q_1 H	0151	R_1 H
:		:		:	
0126	P_{20} L	014E	Q_{20} L	0176	R_{20} L
0127	P_{20} H	014F	Q_{20} H	0177	R_{20} H

Da im Programm 16-Bit-Worte verarbeitet werden sollen, nützt man zweckmäßigerweise die Möglichkeiten dieses Prozessors, mit Registerpaaren zu arbeiten, aus. Das Register A dient als Zähler, der diesmal nur mit dem Hexadezimalwert **14** = dezimal **20** geladen wird, und die Registerpaare BC, HL und DE werden mit den Anfangsadressen der drei Datenfelder geladen.



Da der Akkumulator nicht nur als Zähler, sondern auch als Arbeitsregister Verwendung findet, muß sein Inhalt immer dann, wenn er anschließend für die Datenverarbeitung benötigt wird, durch den Befehl **PUSH PSW** in den Stapelspeicher abgelegt werden, was durch den 5. Befehl geschieht.

Mit dem 6. Befehl wird der Inhalt der Speicherstelle, auf die der Zeiger im Registerpaar BC zeigt, in den Akkumulator geladen und dann mit dem 7. Befehl der Inhalt der Speicherstelle, auf die das Register HL zeigt, dazuaddiert. Ein Carry wird diesmal nicht berücksichtigt, denn wegen der Festlegung der Höchstwerte der Zahlen darf bei der Addition der höherwertigten Byte kein Carry entstehen.

Nach dem Abspeichern des Ergebnisses in die Speicherstelle, auf die das Registerpaar DE zeigt, werden alle Registerpaare incrementiert, und

es erfolgt die Addition der höherwertigen Byte unter Berücksichtigung eines eventuell vorher entstandenen Übertrags. Dann wird auch dieses Ergebnis abgespeichert, und es erfolgt eine erneute Incrementierung aller drei Registerpaare. Bevor jetzt eine Decrementierung des Schleifenzählers erfolgen kann, muß der früher abgespeicherte Inhalt des Akkumulators wieder aus dem Stapelspeicher in den Akkumulator geladen werden, was durch den Befehl **POP PSW** geschieht. Dann erfolgt die Decrementierung des Akkumulators und ein Rücksprung, solange der Inhalt des Akkumulators noch nicht den Wert **00** hat.
Es ist eine gute Übung, dieses Programm so umzustellen, daß der Befehl DAD, durch den der Inhalt zweier Registerpaare addiert wird, zur Verwendung gelangt.

5.10 Programmbeispiel 10

BCD-Addition und -Subtraktion

Lernziele:

Befehle für Dezimalkorrektur. Unterschiedliche Behandlung des Carry-Bit bei der Subtraktion. Prüfen, ob Übertrag oder Borgen entstanden.

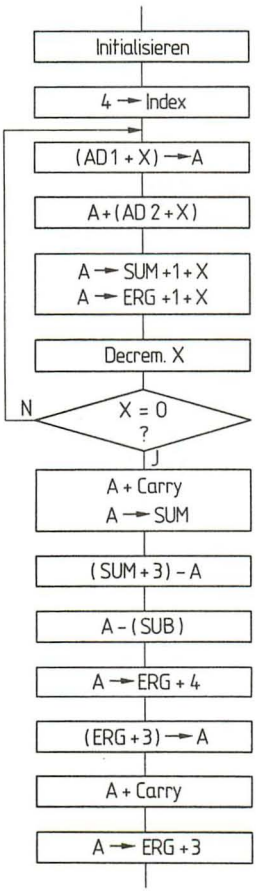
Aufgabe:

Zwei 8stellige Dezimalzahlen sind zu addieren, die Summe ist abzuspeichern. Dann ist eine 2stellige Zahl von der Summe zu subtrahieren, und das Endergebnis ist wieder abzuspeichern. Alle Dezimalzahlen stehen in gepackter BCD-Darstellung und können zwischen **00** *und* **99** *liegen.*

Lösungsweg:

Auch hier gilt es zunächst, die Speicherstellen für die einzelnen Summanden, das Summenergebnis, den Subtrahenden und das Endergebnis festzulegen. Wenn man ein derartiges Programm zum erstenmal entwikkelt, ist es ratsam, zugleich auch ein Zahlenbeispiel aufzuschreiben, mit dessen Hilfe man dann die richtige Arbeit des Programms überprüfen kann (Bild 5.10). Auch bei diesem Programm arbeitet man zweckmäßigerweise mit Indexadressierung für die Additionen, da der gleiche Vorgang viermal ablaufen muß.

Beim zweiten Teil des Programms muß man dann berücksichtigen, daß bei der Addition durchaus ein Übertrag entstehen kann, für den ein




```
AD 1  EQU  0050  \ 1.Summand
      RES     4  / 0050...0053
AD 2  EQU  0054  \ 2.Summand
      RES     4  / 0054...0057
SUM   EQU  005B  \ Summe
      RES     5  / 0058...005C
SUB   EQU  005D    Subtrahend
ERG   EQU  005E  \ Ergebnis
      RES     5  / 005E...0062
```

Beispiel

```
      62  48  53  17   AD 1
+     59  23  65  48   AD 2
  01  21  72  18  65   SUM
-                 87   SUB
  01  21  72  17  18   ERG
```

Speicherbelegung

AD 1	RES 4	0050	62
		0051	48
		0052	53
		0053	17
AD 2	RES 4	0054	59
		0055	23
		0056	65
		0057	48
SUM	RES 5	0058	01
		0059	21
		005A	72
		005B	18
		005C	65
SUB		005D	87
ERG	RES 5	005E	01
		005F	21
		0060	72
		0061	17
		0062	78

Bild 5.10 Flußdiagramm, Speicherzuweisung und Beispiel mit Speicherbelegung für Programmbeispiel 10

10.1 6502

```
 1        CLC              1000  18
 2        SED              1001  F8
 3        LDX    #04       1002  A204
 4 ULLA   LDA    $4F,X     1004  B54F
 5        ADC    $53,X     1006  7553
 6        STA    $58,X     1008  9558
 7        STA    $5E,X     100A  955E
 8        DEX              100C  CA
 9        BNE    ULLA      100D  D0F5
10        LDA    #00       100F  A900
11        ADC    #00       1011  6900
12        STA    $58       1013  8558
13        STA    $5E       1015  855E
14        LDA    $5C       1017  A55C
15        SEC              1019  38
16        SBC    $5D       101A  5D
17        STA    $62       101C  8562
18        LDA    $61       101E  A561
19        SBC    #00       1020  E900
20        STA    $61       1022  8561
21        BRK              1024  00
```

weiteres Speicherbyte reserviert werden muß, so daß für den Speicher der Zwischensumme 5 Byte zu reservieren sind. Das gleiche gilt auch für den Speicher des Endergebnisses, der auch 5 Byte Speicherraum braucht.

6502 Das Programm wird zunächst für den *6502* im Beispiel 10.1 besprochen, da dieser Prozessor sehr einfach auch für die Verarbeitung von Dezimalzahlen im BCD-Format programmiert werden kann. Durch den zweiten Befehl wird das Bit Nr. 3 im Programm-Status-Wort auf **1** gesetzt, wodurch für alle nachfolgenden Befehle die dezimale Arbeitsweise programmiert wurde.

Es folgt das Laden des Indexregisters mit der Anzahl der Schleifen und dann die eigentlichen Arbeitsbefehle mit der Basisadresse, die jeweils der Anfangsadresse des Datenblocks, um den Wert **1** vermindert, entspricht.

Bei diesem Prozessor kann mit Seite-Null-Adressierung gearbeitet werden, so daß nur 2-Byte-Befehle notwendig sind. Das jeweilige Additionsergebnis wird mit dem 6. Befehl in das Datenfeld **SUM** und mit dem 7. Befehl gleichzeitig in das Datenfeld **ERG** abgespeichert.

Durch den 10. Befehl wird der Akkumulator auf **00** gesetzt und durch den 11. Befehl der Wert **00** hinzuaddiert. Da bei dieser Addition zugleich auch ein gesetztes Übertrags-Bit Berücksichtigung findet, zeigt das Ergebnis, ob bei der letzten Addition der höchstwertigsten BCD-Zahl ein Übertrag entstanden ist oder nicht. Das Additionsergebnis wird in die für den Übertrag reservierten Speicherstellen **SUM** und **ERG** abgespeichert.

Nun folgt die Subtraktion. Da bei der Subtraktion ein gesetztes Carry-Bit bedeutet, daß kein Borgen notwendig war, muß vor der Subtraktion das Carry-Bit auf **1** gesetzt werden. Der Akkumulator war vorher mit der letzten Stelle der Summe geladen. Es erfolgt dann mit dem 16. Befehl die Subtraktion, und das Ergebnis wird in die letzte Stelle im Datenblock **ERG** abgespeichert. Jetzt muß geprüft werden, ob eventuell bei der Subtraktion ein Borgen notwendig war. Hierzu wird die vorletzte Stelle des Ergebnisses in den Akkumulator geladen, der Wert **00** unter Berücksichtigung des Carry-Bit subtrahiert und das Ergebnis abgespeichert. Damit ist dieses Programm beendet.

2650

Das im Beispiel 10.2 für den *2650* gezeigte Programm ist im Prinzip in gleicher Weise aufgebaut. Zu berücksichtigen ist allerdings, daß dieser Prozessor keine Adressierung auf Seite Null kennt und nicht durch einfachen Programmbefehl auf dezimale Arbeitsweise programmiert werden kann.

Nach der Initialisierung und dem Laden der Schleifenlänge in das Indexregister 3 wird mit dem vierten Befehl der erste Summand in das Register 0 geladen. Dann muß nach den Regeln der Dezimaladdition, die für diesen Prozessor gelten, der Hexadezimalwert **66** hinzuaddiert werden. Es folgt die Addition des zweiten Summanden und danach der Befehl zur Dezimalkorrektur. Damit ist eine Dezimaladdition ausgeführt, die Ergebnisse werden zweimal abgespeichert und die Schleife erneut durchlaufen.

Im zweiten Teil erfolgt wieder die Berücksichtigung des Übertrags und dann die Subtraktion. Auch bei diesem Prozessor muß das Carry-Bit auf **1** gesetzt werden, wenn kein Borgen erfolgte, was durch den 16. Befehl geschieht. Für die dezimale Subtraktion ist lediglich der Befehl zur

10.2 2650

```
 1        PPSL    H'08'       1000  7708
 2        CPSL    H'01'       1002  7501
 3        LODI,3  H'04'       1004  0704
 4 ULLA   LODA,0  AD1,3-      1006  0F4050
 5        ADDI,0  H'66'       1009  8466
 6        ADDA,0  AD2,3       100B  8F6054
 7        DAR,0               100E  94
 8        STRA,0  SUM+1,3     100F  CF6059
 9        STRA,0  ER6+1,3     1012  CF605F
10        BRNR,3  ULLA        1015  586F
11        EORZ,0              1017  20
12        ADDZ,0              1018  80
13        STRA,0  SUM         1019  CC0058
14        STRA,0  ERG         101C  CC005E
15        LODA,0  ERG+4       101F  0C0062
16        PPSL    H'01'       1022  7701
17        SUBA,0  SUB         1024  AC005D
18        DAR,0               1027  94
19        STRA,0  ERG+4       1028  CC0062
20        LODA,0  ER6+3       102B  0C0061
21        SUBI,0  00          102E  A400
22        DAR,0               1030  94
23        STRA,   ER6+4       1031  CC0061
24        HALT                1034  40
```

Dezimalkorrektur im Anschluß an den eigentlichen Subtraktionsbefehl zu geben. Die Überprüfung, ob geborgt wurde, erfolgt in gleicher Weise wie zum *6502* beschrieben.

6800 Beim Programm für den *6800*, das im Beispiel 10.3 gezeigt wird, muß wieder berücksichtigt werden, daß dieser Prozessor ein 16-Bit-Indexregister hat. Der Prozessor kann auch nicht generell für dezimale Arbeitsweise programmiert werden, und die Dezimalkorrektur ist anders durchzuführen als beim *2650*.

Die Additionsschleife ist genauso aufgebaut, wie es im Beispiel 9.3

10.3 6800

```
 1        LDX     #$0050   1000  CE0050
 2 ULLA   LDAA    $03,X    1003  A603
 3        ADCA    $07,X    1005  A907
 4        DAA              1007  19
 5        STAA    $0C,X    1008  A70C
 6        STAA    $12,X    100A  A712
 7        DEX              100C  09
 8        CPX     #$004C   100D  8C004C
 9        BNE     ULLA     1010  26F1
10        LDAA    #00      1012  8600
11        ADCA    #00      1014  8900
12        STAA    SUM      1016  9758
13        STAA    ERG      1018  975E
14        LDAA    #$9A     101A  869A
15        SUBA    SUB      101C  905D
16        ADCA    ERG+4    101E  9962
17        DAA              1020  19
18        STAA    ERG+4    1021  9762
19        BCS     END      1023  2507
20        LDAA    #$99     1025  8699
21        ADCA    ERG+3    1027  9961
22        DAA              1029  19
23        STAA    ERG+3    102A  9761
24 END    WAI              102C  3E
```

erklärt wurde. Nach dem eigentlichen Additionsbefehl muß für die Dezimalkorrektur der Befehl **DAA** gegeben werden.

Mit dem Befehl 10 bis 13 wird ein eventueller Übertrag berücksichtigt.

Die dezimale Subtraktion muß bei diesem Prozessor nach einer besonderen Methode vorgenommen werden, wie im Abschnitt 2.5 beschrieben ist. Zunächst wird in das Arbeitsregister, hier den Akkumulator A, der Wert **9A** geladen. Dann wird der Subtrahent vom Akkumulator subtrahiert und der Minuend zum Akkumulator hinzuaddiert. Es folgt der

10.4 8080

```
 1        LXI     B,0053H 1000  015300
 2        LXI     H,0057H 1003  215700
 3        LXI     D,005CH 1006  115C00
 4 ULLA   POP     PSW     1009  F1
 5        LDAX    B       100A  0A
 6        ADC     M       100B  8E
 7        DAA             100C  27
 8        STAX    D       100D  12
 9        DCX     B       100E  0B
10        DCX     H       100F  2B
11        DCX     D       1010  1B
12        PUSH    PSW     1011  F5
13        MVI     A,58H   1012  3E58
14        CMP     E       1014  BB
15        JNZ     ULLA    1015  C20910
16        POP     PSW     1018  F1
17        MVI     A,00    1019  3E00
18        ACI     A,00    101B  CE00
19        STAX    D       101D  12
20        LXI     B,005EH 101E  015E00
21 RITA   LDAX    D       1021  1A
22        STAX    B       1022  02
23        INX     D       1023  13
24        INX     B       1024  03
25        MVI     A,63H   1025  3E63
26        CMP     E       1027  BB
27        JNZ     RITA    1028  C22110
28        LXI     H,005DH 102B  215D00
29        MVI     A,9AH   102E  3E9A
30        SBB     M       1030  9E
31        LXI     H,0062H 1031  216200
32        ADC     M       1034  8E
33        DAA             1035  27
34        MOV     M,A     1036  77
35        JC      END     1037  DA4010
```

```
36          DCX     H         103A  2B
37          MVI     A,99H     103B  3E99
38          ADC     M         103D  8E
39          DAA               103E  27
40          MOV     M,A       103F  77
41  END     HLT               1040  76
```

Befehl zur Dezimalkorrektur, und das Ergebnis kann dann als Dezimalwert abgespeichert werden. Ist bei dieser Operation ein Carry-Bit gesetzt worden, so bedeutet dies bei der BCD-Subtraktion, daß kein Borgen notwendig gewesen ist, und es kann ein Sprung zum Ende des Programms erfolgen. Wurde das Carry-Bit nicht gesetzt, so ist in der nächsthöheren Speicherstelle der Wert **1** zu subtrahieren, was durch Einsetzen von **99** statt **9A** erfolgt. Eine Subtraktion ist dann nicht mehr notwendig.

Das gleiche Programm, für den *8080* geschrieben, das im Beispiel 10.4 dargestellt ist, ähnelt im ersten Teil dem Programm im Beispiel 9.4, da auch hier wieder die drei Registerpaare als Zeiger für die beiden Summandenadressen und die Ergebnisadresse Verwendung findet. Hier wird allerdings nicht das Register A als Schleifenzähler verwendet, sondern es wird der Inhalt des Registers E als Kriterium für den Schleifenabbruch abgefragt. **8080**

Da aber beim **CMP**-Befehl das Carry-Bit beeinflußt wird, muß vor dem Vergleich das **PSW**, das den Übertrag der Addition enthält, im Stapelspeicher gerettet und vor der nächsten Schleife wieder zurückgeschrieben werden.

Die Dezimalkorrektur erfolgt beim *8080* auf gleiche Weise, wie für den *6800* beschrieben, und zwar sowohl was Addition als auch Subtraktion betrifft.

Da nur 3 Registerpaare als Zeiger zur Verfügung stehen, muß nach dem Additionsteil des Programms noch zusätzlich der Inhalt des Datenfeldes **SUM** in das Datenfeld **ERG** überschrieben werden, wofür die Befehle 20 bis 26 verwendet werden. Das Registerpaar BC wird als Zeiger verwendet, der zunächst auf die erste Adresse des Datenblocks **ERG** zeigt. Am Ende der 1. Schleife zeigt das Register DE auf die erste Adresse des Datenblocks **SUM.** Mit dem 21. Befehl wird nun der Inhalt, auf den das Registerpaar DE zeigt, in den Akkumulator geladen und mit dem 22. Befehl in die Speicherstelle abgespeichert, auf die **BC** zeigt. Dann werden

beide Register incrementiert, und die Schleife wird erneut durchlaufen, so lange, bis das Register E den Wert **63** hat, womit angezeigt wird, daß der letzte Speicherplatz im Datenfeld **ERG** bearbeitet wurde. Der letzte Teil des Programms vom Befehl 32 an erledigt die Subtraktion in gleicher Weise, wie es im Beispiel 10.3 für den *6800* beschrieben wurde.

6
Struktur und Befehlsaufbau einiger weiterer Mikroprozessor-Typen

Es gibt eine Reihe von weiteren Mikroprozessoren, deren innerer Aufbau und Befehlssätze den bisher besprochenen Typen ähnelt und die darum nicht im einzelnen besprochen werden sollen.

Obwohl die beschriebenen «Standardprozessoren» durchaus in der Lage sind, alle nur denkbaren Aufgaben zu erfüllen und darum wohl auch noch eine ganze Reihe von Jahren im Programm der Hersteller bleiben werden, zeichnen sich doch schon neue Entwicklungen ab, auf die hier noch eingegangen werden soll.

- für kleinere Aufgaben sind Ein-Chip-Mikrocomputer entwickelt worden, bei denen alle Bauelemente, die für ein komplettes Mikroprozessor-System notwendig sind, in einem Baustein, also in einem Gehäuse, vereinigt wurden. Als Beispiel wird hier der Mikroprozessor-Typ 8048 besprochen.
- Für Anwender, die schon eine gewisse Erfahrung mit Standard-Mikroprozessoren besitzen, wurden sehr leistungsstarke, in ihrem Aufbau aber auch recht komplexe Prozessoren geschaffen, die mit einem sehr umfangreichen Befehlssatz und schneller Arbeitsgeschwindigkeit auch für Aufgaben geeignet sind, die bisher den Minicomputern vorbehalten waren. Als Beispiel wird hier der 8-Bit-Prozessor Z 80 und der 16-Bit-Prozessor 9900 behandelt.

Um den Umfang dieses Buches nicht zu sprengen, werden allerdings nur die Besonderheiten dieser Prozessoren erläutert. Die Befehlssätze werden nur insoweit besprochen, als darin noch Befehle vorkommen, deren Wirkungsweise vorher noch nicht erklärt wurde.

6.1 Der Ein-Chip-Mikrocomputer 8048

Es zeigt sich immer mehr, daß Mikroprozessoren auch für verhältnismäßig einfache Steuerungsaufgaben gut verwendbar sind. Für diesen Einsatz werden meist keine großen Speicher benötigt, aber es kommt darauf an, daß die ganze Steuerung kompakt und damit billig aufzubauen ist.

Eine Lösung für derartige Aufgabenstellungen sind Ein-Chip-Mikrocomputer, bei denen die CPU und jeweils eine Anzahl von ROM- und RAM-Speicherplätzen zusammen mit dem Taktgenerator und gegebenenfalls noch weiteren Steuerungselementen so auf einer hochintegrierten elektronischen Schaltung vereinigt sind, daß sie in einem Bausteingehäuse untergebracht werden können. Als Beispiel für diese Gruppe von Bauelementen sollen hier die Besonderheiten der Programmierung für den *8048*, die auch für die anderen Mitglieder seiner Typenfamilie gelten, besprochen werden.

Ein vereinfachtes Blockschaltbild des *8048* und die bei ihm durch Programmbefehle zugänglichen Register zeigt das Bild 6.1.

Es ist zu erkennen, daß für die Programmspeicherung 1 KByte und für die Datenspeicherung 64 Byte zur Verfügung stehen. 27 Ein- bzw. Ausgabeleitungen können angesprochen werden, und außer den Bauelementen für die Takterzeugung befindet sich auf dem Chip noch ein *Zeitgeber/Zähler* (TIMER), der vom Programmierer mitverwendet werden kann.

Der Prozessor besitzt neben einem Akkumulator zwei Registerbänke mit jeweils 8 Hilfsregistern, die wahlweise durch Programmbefehle aufgerufen werden können. Es gibt ein Programm-Status-Register, das auch einen Stapelzeiger enthält, und ein Programmadreß-Register (PC) mit einer Kapazität von 12 Bit, mit dem bis zu 4096 Adressen angesprochen werden können. Bei voller Ausnutzung der Kapazität sind allerdings zusätzliche Speicherbausteine notwendig.

Es gibt verschiedene Ausführungen dieses Prozessors, die sich bezüglich der Speicheranzahl und der Speicherausführung unterscheiden, worauf hier aber nicht weiter eingegangen werden soll.

Wesentlich anders als bei den bisher besprochenen Mikroprozessor-Typen ist der Aufbau des Befehlssatzes, der sehr viele Ein-Byte-Befehle, einige Zwei-Byte-Befehle und keine Drei-Byte-Befehle verwendet. Die Beschränkung auf Ein- und Zwei-Byte-Befehle ist dadurch möglich, daß

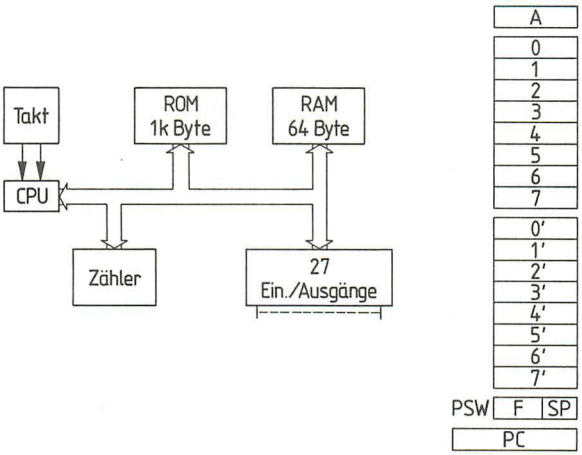


Bild 6.1 Blockschaltbild und Registeraufbau bei den Prozessoren der 8048-Familie

der gesamte adressierbare Speicherbereich in zwei Adreß-Bänke eingeteilt ist. Im normalen Programmlauf arbeitet der Prozessor nur innerhalb einer Adreßbank, das heißt also entweder in der Bank **0** von der Adresse **000** bis zur Adresse **7FF** oder in der Bank **1** von **800** bis **FFF.** Der Übergang von einem Adreßbereich zum anderen erfolgt durch die Steuerbefehle **SEL MB0** bzw. **SEL MB1.**

Durch die ähnlichen Steuerbefehle **SEL RB0** bzw. **SEL RB1** kann auch jeweils bestimmt werden, ob mit den Registern der Registerbank **0** oder den Registern der Registerbank **1** gearbeitet werden soll. Der Adressenbereich der ROM- und RAM-Speicher und die Lage der Adreß- und Register-Bänke sind im Bild 6.2 dargestellt.

Der *8048* kommt mit nur wenigen Adressierarten aus. Außer der Register-Adressierung mit Ein-Byte-Befehlen für alle Operationen, bei denen der Akkumulator allein oder der Akkumulator und ein anderes Register beteiligt sind, gibt es auch Zwei-Byte-Befehle mit unmittelbarer Adressierung, bei denen der zweite Operand im zweiten Befehls-Byte steht.

Der *8048* kennt auch eine indirekte Adressierung für die RAM-Speicherplätze. In diesem Fall wird das Register 0 oder das Register 1 je nach Programmierung im Befehl als Zeiger für die Operandenadresse verwen-

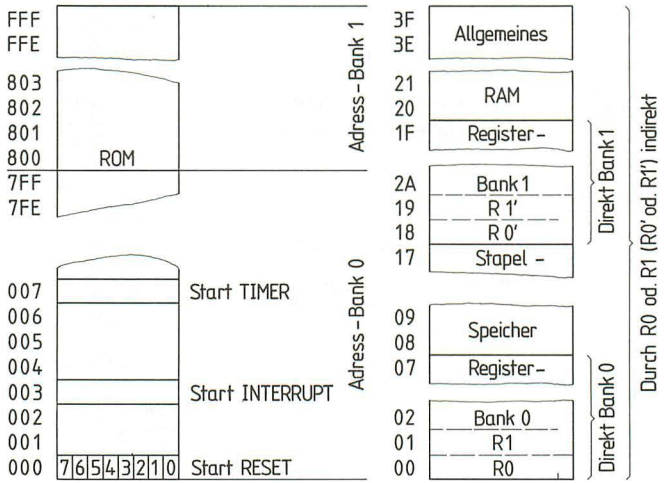


Bild 6.2 ROM- und RAM-Adressierbereiche bei den Prozessoren der 8048-Familie

det. Wenn nur das RAM auf dem Chip verwendet wird, genügen für die dort vorhandenen 64 Speicherplätze die 6 niederstwertigen Bit des Zeigers. Beim Anschluß externer Speicher stehen dann nochmal der durch 8 Bit adressierbare Speicherraum, also 256 Byte, für RAM-Speicher mit indirekter Adressierung zur Verfügung.

Alle Datentransport-Befehle haben den Akkumulator als Datenziel bzw. Datenquelle. Außer den **MOV**-Befehlen, die ähnlich arbeiten wie beim *8080*, gibt es noch Austausch-Befehle mit dem Kürzel **XCH**, mit denen der Inhalt von Registern oder RAM-Speicherstellen mit dem Inhalt des Akkumulators ausgetauscht werden kann.

Die Datenverarbeitung findet bei diesem Prozessor auch grundsätzlich im Akkumulator statt. Der *8048* kann Additionen mit und ohne Berücksichtigung des Carry-Bit ausführen, und es können die logischen Verknüpfungen «UND», «ODER» bzw. «Exklusiv ODER» ausgeführt werden. Der Prozessor kennt allerdings keine Subtraktionsbefehle. Subtraktionen müssen gegebenenfalls durch Kombination anderer Befehle in einem Unterprogramm extra programmiert werden.

Als Datenverarbeitungs-Befehle mit einem Operanden kann der Prozessor den Inhalt des Akkumulators rechts und links wahlweise unter

Einschluß des Carry-Bit rotieren, incrementieren, decrementieren, löschen und komplementieren. Auch die Inhalte der einzelnen Register und die indirekt adressierten RAM-Speicherstellen können durch Ein-Byte-Befehle incrementiert, Registerinhalte auch decrementiert werden.

Es stehen weiterhin Befehle für die Dezimalkorrektur des Akkumulator-Inhalts und für das Vertauschen der beiden Halbbyte im Akkumulator zur Verfügung (SWAP).

Für Ein- und Ausgabeoperationen stehen zwei auf Sockelstifte herausgeführte Port, die PORT 1 und PORT 2 genannt werden, sowie der gleichfalls auf Sockelstifte gelegte System-Bus des Prozessors zur Verfügung. Hiermit können insgesamt 24 Ein- oder Ausgabeleitungen bedient werden. Es gibt außerdem noch drei Eingänge für Einzelsignale (**T0, T1** und **INT**), die für bedingte Sprungbefehle direkt abgefragt werden können. Hierdurch sind Programmverzweigungen möglich, ohne daß der Inhalt eines Eingabeports in den Akkumulator geladen werden muß.

Mit den Eingabebefehlen **IN** und den Ausgabebefehlen **OUTL** wird jeweils der Inhalt des angesprochenen Port in den Akkumulator geladen bzw. der Inhalt des Akkumulators an das Port ausgegeben. Es gibt aber auch noch die Befehle **ANL** und **ORL** mit unmittelbarer Adressierung. Durch sie wird die logische UND- bzw. ODER-Verknüpfung des Port-Inhalts mit der Maske im zweiten Befehls-Byte durchgeführt.

Beim *8048* gibt es nur einen Sprungbefehl, **JMP,** und einen Befehl zum Anruf von Unterprogrammen, **CALL,** mit dem der gesamte Adreßbereich absolut in der gerade bearbeiteten Adreß-Bank erreicht werden kann. Es handelt sich hier auch um Zwei-Byte-Befehle. Da aber für die Adressierung des Adreßraums einer Bank 11 Bit benötigt werden, reichen die 8 Bit des zweiten Befehls-Byte nicht aus. Die drei höchstwertigen Bit der Sprungziel-Adresse stehen darum bei diesen Befehlen in den drei höchstwertigen Bit des ersten Befehls-Byte.

Wenn innerhalb des vollen Adressenbereichs über die Grenze einer Adreß-Bank hinaus gesprungen werden soll, muß zunächst ein Befehl zum Bankwechsel gegeben werden. Das Umschalten auf die zweite Bank erfolgt erst, wenn der Sprungbefehl ausgeführt wird. Sobald eine bestimmte Bank angewählt worden ist, erfolgen alle weiteren Sprünge innerhalb dieser Bank, bis ein neuer Befehl zum Bankwechsel gegeben wird. Ein Unterprogramm in der jeweils anderen Bank kann nur angerufen werden, wenn vor dem entsprechenden Befehl wieder ein Befehl zum Bankwechsel gegeben wurde.

Bedingte Sprungbefehle erlauben nur einen Sprung in eine Adresse auf der gerade durch das Programm benutzten Seite. Im zweiten Byte dieser Befehle werden die 8 niederstwertigen Bit der Sprungzieladresse angegeben.

Für die Entscheidung, ob gesprungen werden soll, kann das Carry-Bit im Programm-Status-Register abgefragt oder geprüft werden, ob der Akkumulator den Inhalt Null hat.

Bedingte Sprünge können auch so programmiert werden, daß die Signale der Eingänge **T0, T1,** oder **INT** als Sprungbedingungen abgefragt werden. Weiterhin ist es möglich, bedingte Sprünge durch die Ausgänge des auf dem Chip enthaltenen Zeitgebers zu veranlassen.

Der Befehl **DJNZ** ist bei Schleifen sehr nützlich, da der Sprung immer so lange stattfindet, wie das angezogene Register noch nicht den Inhalt Null hat. Beim Befehl **JBB** erfolgt der Sprung immer dann, wenn ein im Befehl angezogenes Bit im Akkumulator gesetzt ist.

Für die Bedienung des Zeitgeber/Zählers gibt es eine Reihe von Befehlen, mit denen dieser geladen, ausgelesen, gestartet und gestoppt werden kann.

Das Programm-Status-Register ist wie im Bild 6.3 gezeigt aufgebaut. Die Bit Nr. 6 und 7 zeigen in üblicher Weise die Ergebnisse von Datenverarbeitungs-Operationen. Das Bit Nr. 5 kann durch Programmbefehle gesetzt oder gelöscht werden und gilt dann auch als Entscheidungsmerkmal für bedingte Sprungbefehle. Das Bit Nr. 4 dient zur Wahl der zu verwendeten Registerbank, und die Bit Nr. 0 bis 2 stellen den Zeiger für den Stapelspeicher dar.

Der im RAM-Bereich angeordnete Stapelspeicher ist wie im Bild 6.4 gezeigt aufgebaut. Er dient zum Speichern der Rücksprungadressen nach dem Anruf eines Unterprogramms, wobei die Bedienung des Stapelspeichers selbsttätig, ohne daß sie programmiert werden muß, nach den **CALL-** bzw. **RETR**-Befehlen erfolgt. Es werden jeweils 12 Adreß-Bit und die oberen 4 Bit des Programm-Status-Wortes in zwei RAM-Byte abgespeichert und danach der Stapelzeiger um den Wert **1** erhöht, womit er auf die nächste freie Stapelstelle zeigt.

Befehle für die Bedienung des Stapelspeichers gibt es also beim *8048* nicht. Die übrigen Befehle für das Setzen und Löschen einzelner Bit im Programm-Status-Register und für das Sperren und Freigeben des Interrupt entsprechen im wesentlichen den Befehlen bei den anderen Prozessoren.

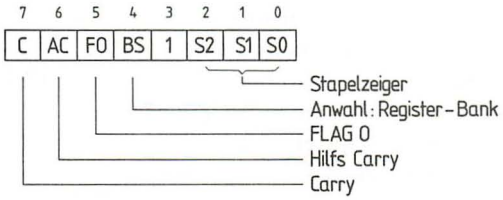


Bild 6.3 Aufbau des Programm-Status-Worts bei den Prozessoren der 8048-Familie

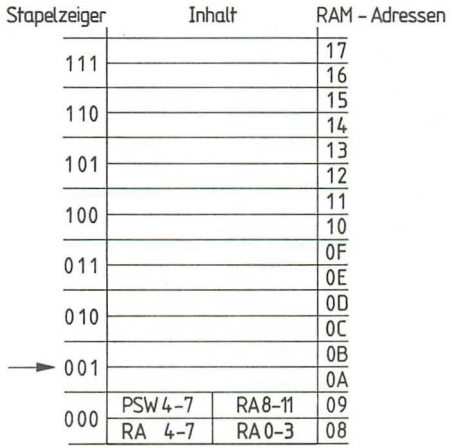


Bild 6.4 Aufbau des Stapelspeichers für Rücksprungadressen im RAM des 8048

Bei der Programmierarbeit für den *8048* muß man immer die Grenzen der Seiten und die Grenzen der Bänke im Auge behalten. Wenn alle Regeln beachtet werden, bleiben die Programme kurz, da viele Operationen mit Ein-Byte-Befehlen veranlaßt werden können. Die Möglichkeit, auch 1-Bit-Informationen auf einfache Weise zu verarbeiten, machen den *8048* insbesondere für Aufgaben in der Steuerungstechnik interessant, während er für Rechenoperationen und umfangreiche Datenverarbeitung nicht so geeignet erscheint.

Das Hauptverwendungsgebiet für den *8048* und die anderen Prozessoren seiner Familie dürfte in erster Linie bei Aufgaben liegen, bei denen

genügend große Stückzahlen zu erwarten sind, die eine Maskenprogrammierung für das auf dem Chip befindliche ROM erlauben. Es gibt zwar von diesem Prozessor auch Ausführungen mit einem EPROM als Programmspeicher, jedoch erscheinen für kleinere Serien, bei denen man gern PROM-Bausteine als Speicherelemente verwendet, die im Kapitel 4 und 5 behandelten Universal-Prozessoren etwas besser geeignet.

6.2 Der 8-Bit-Mikroprozessor Z 80

Wenn man den Aufbau der durch Programmbefehle zugänglichen Register des *Z 80* betrachtet (Bild 6.5), erkennt man auf den ersten Blick die große Ähnlichkeit mit den Prozessoren der *8080/85*-Familie. Auch der Z 80 besitzt die zu Registerpaaren zusammenzufassenden Hilfsregister B, C, D, E, H und L sowie den darüberstehenden Akkumulator. Er entspricht also hier genau dem *8080*. Außerdem gibt es aber noch eine zweite Registerbank, in der die gleichen Register als A', B', C', D', E', H' und L' vorhanden sind. Zusätzlich hat der *Z 80* dann noch ein zweites Programm-Status-Register und die zwei Indexregister X und Y mit je 16 Bit.

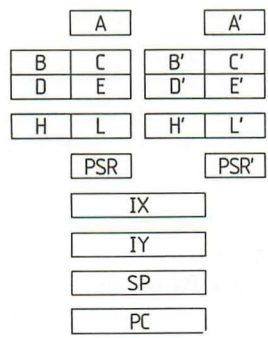


Bild 6.5 Durch Programmbefehle zugängliche Register beim Z 80

Durch diesen internen Aufbau wird das ganze Konzept des *Z 80* bestimmt. Er ist in der Lage, alle Befehle, die ihm in der Maschinensprache des *8080* gegeben werden, auszuführen. Darüber hinaus gibt es aber bei diesem Prozessor noch sehr viel mehr mögliche Operationen, die

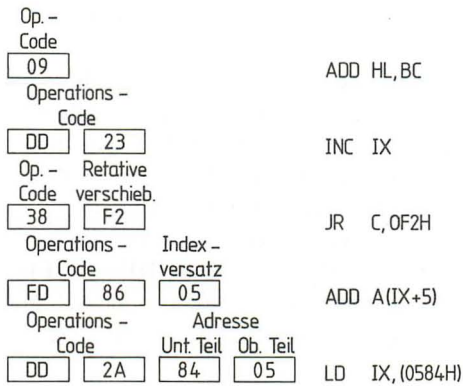


Bild 6.6 Aufbau der verschiedenen Befehle des Mikroprozessors Z 80

durch eine große Anzahl von Befehlen, die nicht im Befehlssatz des *8080* enthalten sind, veranlaßt werden können.

Verwendet wird hier die Tatsache, daß einige 8-Bit-Operationscodes im Befehlssatz des *8080* überhaupt nicht benutzt werden. Der *Z 80* verwendet nun diese unbenutzten Bitkombinationen als erstes Befehls-Byte für seine zusätzlichen Befehle. Um möglichst viele verschiedene Befehle in diesen insgesamt nur 12 freien Bitmustern unterbringen zu können, wird bei den Zusatzbefehlen meist das zweite Byte auch noch für den Operationscode verwendet, so daß sich dadurch Befehlsformate bis zu vier Byte Länge ergeben (Bild 6.6).

Durch diese Zusatzbefehle werden nun sehr viele Programmiermöglichkeiten für den *Z 80* geschaffen, die es beim *8080* nicht gibt. Es würde den Umfang dieses Kapitels sprengen, wenn alle Möglichkeiten ausführlich beschrieben würden, aber die wesentlichen Erweiterungen des Befehlssatzes, die dieser Prozessor bietet, sollen doch im folgenden gezeigt werden.

Wenn auch die Syntax der *Z 80*-Assemblersprache im wesentlichen mit derjenigen des *8080* übereinstimmt, so gibt es doch einige Unterschiede bei den mnemonischen Kürzeln. Statt **MOV, STAX** und **LDAX** steht beim *Z 80* einheitlich für alle Datentransport-Befehle das Kürzel **LD.** Statt **ORA** steht nur **OR** usw. Das sind aber nur unwesentliche Unterschiede. Die vielseitigen Verwendungsmöglichkeiten dieses Pro-

zessors kommen nur durch die zusätzlich möglichen Befehle zum Ausdruck.

Zunächst gibt es eine Reihe von Austauschbefehlen mit dem Kürzel **EX,** durch die die Informationen, die in den Registerbänken 0 bzw. 1 stehen, ausgetauscht werden. Mit Hilfe dieser Befehle ist es möglich, Registerinhalte beim Einsprung in Unterprogramme zu retten, ohne daß der Stapelspeicher benutzt zu werden braucht.

Mit seinen zwei 16-Bit-Indexregistern, auf die durch eine ganze Reihe von Befehlen mit dem Operationscode **DE** bzw. **FD** im ersten Befehls-Byte zugegriffen werden kann, können praktisch alle Operationen mit Index-Adressierung über eines der beiden Indexregister ausgeführt werden.

Insbesondere für Aufgaben in der Steuerungstechnik ist es interessant, daß es beim *Z 80* eine ganze Reihe von Befehlen gibt, die nur einzelne Bit-Positionen in den verschiedenen Registern betreffen.

Durch eine Reihe von Befehlen mit dem Kürzel **BIT** werden die einzelnen Bit-Positionen in den verschiedenen Registern geprüft. Je nachdem, ob dort eine **1** oder eine **0** steht, wird in die Zero-Position des Programm-Status-Registers eine **0** oder eine **1** als Ergebnis der Prüfung gesetzt.

Es gibt weiterhin eine Reihe von Vergleichsbefehlen, bei denen zusätzlich zum Vergleich das Vergleichsregister noch decrementiert oder incrementiert wird. Durch diese Befehle wird der Aufbau von Schleifen erleichtert, wie es bei der Besprechung des *2650* erklärt wurde.

Außer den Ein- und Ausgangsoperationen, die über den Akkumulator laufen, erlaubt der *Z 80* auch Ein- und Ausgaben in und aus allen anderen Registern, wobei bei einigen Befehlen zugleich noch eine Incrementierung bzw. Decrementierung von Registern vorgenommen wird.

Es gibt eine Reihe von zusätzlichen unbedingten Sprungbefehlen, bei denen das Sprungziel dem Inhalt eines der Indexregister entspricht.

Der *Z 80* kennt auch einige Sprungbefehle mit relativer Adressierung, die es beim *8080* nicht gibt. Bei ihnen steht im zweiten Befehls-Byte, wie auch sonst üblich, die relative Verschiebung von **−128** und **+127** als Zweier-Komplementzahl.

Außer den Rotationsbefehlen kennt der *Z 80* auch Befehle für arithmetisches und logisches Schieben für die Inhalte aller Register.

Diese kurze Zusammenfassung zeigt, daß der *Z 80* einen außerordentlich vielseitigen Befehlssatz hat, in dem praktisch alle Befehlsausfüh-

rungsmöglichkeiten, die andere Prozessortypen bieten, zusammengefaßt sind. Man kann mit dem *Z 80* Programme aufbauen, die ähnlich arbeiten wie auf einem *8080*. Ebenso kann man aber auch für diesen Prozessor Programme schreiben, die dem Programmaufbau z.B. für den *2650* oder *6502* entsprechen. Ein Programmierer, der alle Möglichkeiten dieses Prozessors ausnutzt, muß sein Handwerk wirklich verstehen.

6.3 Der 16-Bit-Mikroprozessor 9900

Alle bisher besprochenen Mikroprozessoren arbeiten nach der sogenannten *Speicher-zu-Register-Architektur,* wobei die *Register-zu-Register-Architektur* mit eingeschlossen ist. Man versteht darunter, daß bei Datenverarbeitungs-Befehlen, bei denen zwei Operanden beteiligt sind,

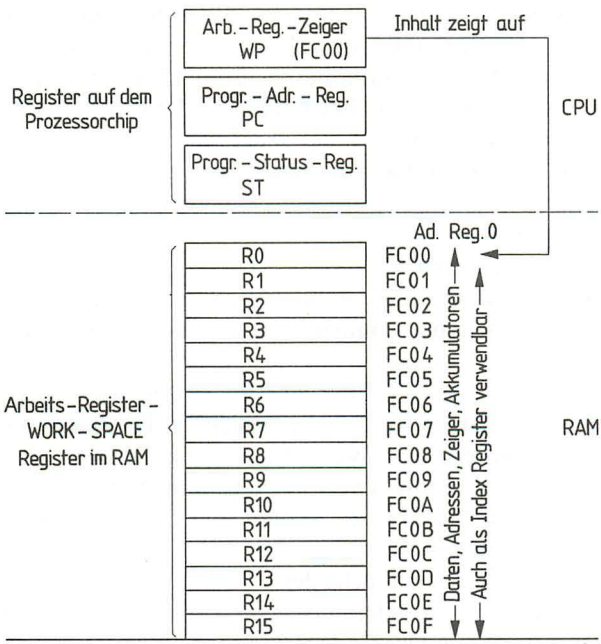


Bild 6.7 Aufbau der Register für die Mikroprozessoren der 9900-Familie

mindestens einer der Operanden in einem Register stehen muß, während der andere aus einer Speicherstelle geholt werden kann.

Demgegenüber arbeitet der *9900* nach der *Speicher-zu-Speicher-Architektur,* was bedeutet, daß auch beide Operanden vor der Befehlsausführung in Speicherstellen außerhalb der Zentraleinheit stehen können. Die eigentliche Datenverarbeitung findet natürlich auch bei diesem Prozessor in der CPU statt. Die Daten werden also von den Speicherstellen in die CPU transportiert, dort verarbeitet, und das Ergebnis wird dann wieder in der im Befehl festgelegten Speicherstelle abgelegt.

Diese Arbeitsweise bestimmt auch den Aufbau des 9900, der, wie Bild 6.7 zeigt, auf dem Prozessor-Chip nur noch drei durch Programmbefehle zugängliche 16 Bit breite Register hat. Eines davon ist das übliche Programmadreß-Register, dann gibt es auch ein Programm-Status-Register, am bemerkenswertesten ist aber das dritte Register, das WORK SPACE POINTER oder Arbeitsregister-Zeiger genannt wird.

In den Arbeitsregister-Zeiger kann durch Programmbefehle eine beliebige 16-Bit-Adresse geladen werden, an die hardwaremäßig ein RAM-Speicher angeschlossen ist. Die durch den Wert im Arbeitsregister-Zeiger gekennzeichnete Adresse sowie die nächsten 15 in aufsteigender Reihenfolge stehenden Adressen bilden dann von Register 0 bis Register 15 den eigentlichen Arbeitsbereich des Prozessors.

Durch Änderung des Inhalts im Arbeitsregister-Zeiger kann man also sehr einfach mit einer beliebig großen Anzahl von Registerbänken arbeiten und bei ineinander verschachtelten Programmteilen gegebenenfalls die Registerinhalte retten.

Die Befehle des *9900* sind grundsätzlich wortweise, das heißt in 16-Bit-Abschnitten, aufgeteilt und können eine Länge von einem, zwei oder drei Worten haben.

Etwas verwirrend ist es, daß der *9900* die einzelnen Bit in einem Bitmuster anders numeriert, als es sonst üblich ist. Die Numerierung der Bitpositionen in einem Wort beginnt hier ganz links mit 0 und endet rechts mit 15, wie es in Bild 6.8 zu sehen ist. Ein Wort kann durch vier Hexadezimalzahlen beschrieben werden und dann auch schon einen vollständigen Befehl darstellen.

Auch bei der Assemblersprache des *9900* muß man etwas umdenken. Während bei den meisten anderen Assemblersprachen, wie zum Beispiel beim *8080* und *Z 80,* im Operandenfeld zuerst links das Datenziel bzw. das Ergebnisregister steht, wodurch sich, bildlich gesprochen, ein Daten-

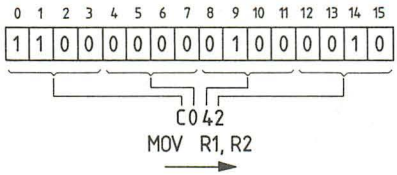


Bild 6.8 Bit-Numerierung und Befehlsdarstellung beim 9900

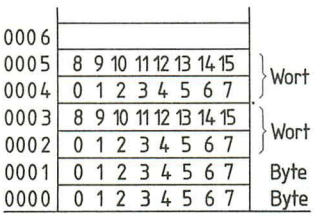


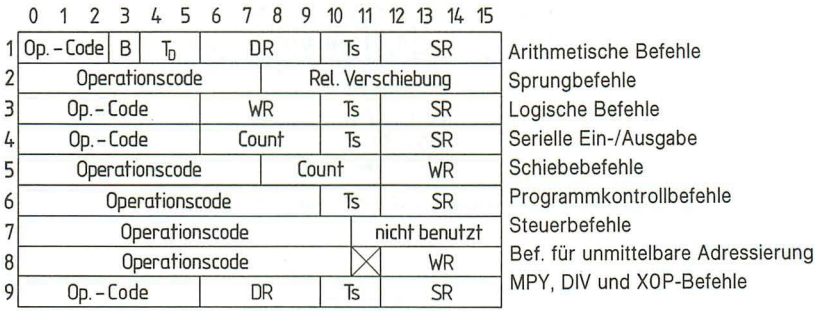
Bild 6.9 Speicherbelegung für Byte und Wort beim 9900

transport von rechts nach links ergibt, steht beim *9900* im Operandenfeld zuerst die Datenquelle und dann das Datenziel oder der Speicherplatz für das Ergebnis.

Als 16-Bit-Prozessor kann der *9900* auch 16 Bit breite Datenworte mit einem Befehl verarbeiten. Da für viele Aufgaben aber auch nur 8-Bit-Informationen ausreichen, hat er die Möglichkeit einer Byte-Verarbeitung, wodurch der Speicherraum dann besser ausgenutzt wird. Die an diesen Prozessor anzuschließenden Speicherbausteine müssen darum auch in Byte organisiert und adressierbar sein. Wie Bild 6.9 zeigt, nimmt ein Wort die Stelle von zwei Speicherplätzen ein. Ist der Prozessor auf Wort-Arbeitsweise programmiert, werden nur die Speicherplätze mit gerader Adresse angesprochen, in die dann der untere Teil des zu speichernden Wortes abgelegt wird. Der obere Teil kommt dann in die nächstfolgende Speicheradresse.

Durch Setzen oder Löschen eines bestimmten Bit im Operationscode des Befehls kann der Programmierer beliebig von der Arbeitsweise in Byte-Struktur zur Arbeit in Wort-Struktur wechseln.

Der *9900* kennt 69 verschiedene Befehle, die durch ein bis vier Buchstaben lange mnemonische Kürzel ausgedrückt werden. Da für das erste Befehlswort, das wie üblich den Operationscode enthält, 16 Bit zur Ver-



B - Byte-Indikator (1 = Byte)
T_D - Adreßtyp für Ziel
DR - Zielregister Nr.
Ts - Adreßtyp für Quelle
SR - Quell-Register Nr.
Count - E/A- oder Schiebezahl (1...16)
WR - Arbeitsregister Nr.

Assember-Symb.	T_D oder Ts	Adressierarten
	00	Register
*	01	Register Indirekt
@	10	Direkt (Absolut) SR od. DR = 000
(R)	10	Index, SR od. > 0
*+	11	Register Indirekt mit Auto-Increment

Bild 6.10 Formate des ersten Befehlsworts beim 9900

fügung stehen, lassen sich darin eine sehr große Anzahl von Variationen für die einzelnen Grundbefehle unterbringen, so daß insgesamt über 500 verschiedene Befehle möglich sind.

Auf den ersten Blick erscheint dies sehr verwirrend. Wenn man sich jedoch etwas näher mit dem Befehlsaufbau befaßt, erkennt man die strenge Logik, die dem Befehlssatz zugrunde liegt. Wie Bild 6.10 zeigt, gibt es für das erste Befehlswort neun verschiedene Formate, wobei man mit fünf Adressierarten arbeiten kann.

Da der Prozessor mit der Speicher-zu-Speicher-Architektur arbeitet, braucht man die sonst bei der Mikroprozessor-Programmierung sehr oft

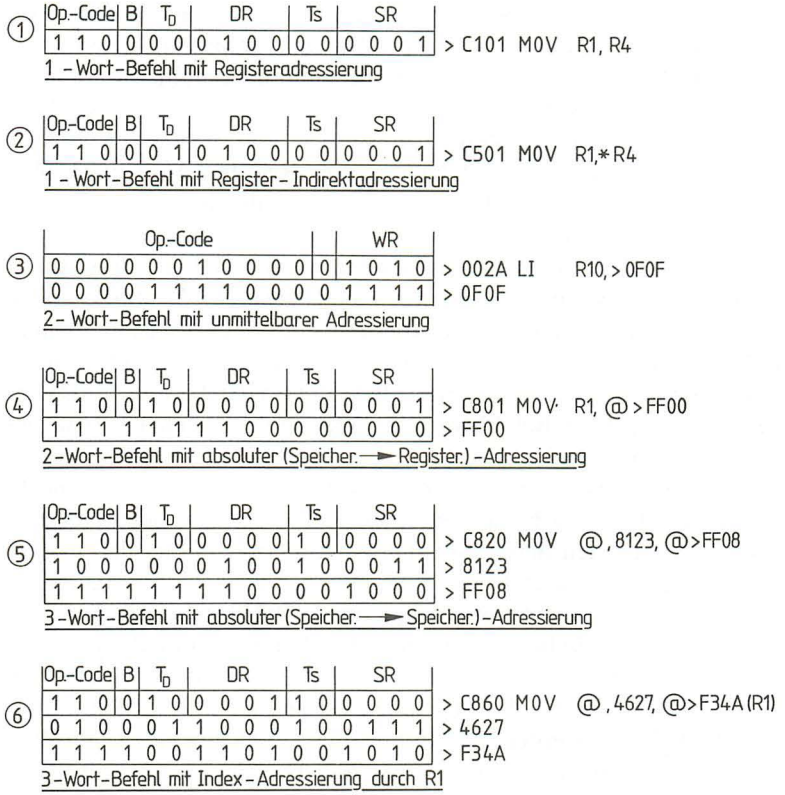


Bild 6.11 Beispiele für den Befehlsaufbau beim 9900

notwendigen Datentransport-Befehle viel seltener. Weil alle Register wahlweise als Akkumulator, als Indexregister, als Zeiger für indirekte Adressierung und natürlich auch als Zwischenspeicher für Daten und Adressen verwendet werden können, ergibt sich ein sehr effektiver Programmaufbau.

Die fünf möglichen Adressierarten können praktisch bei allen Befehlen, bei denen sie sinnvoll sind, angewandt werden. Eine gewisse Vereinfachung gegenüber 8-Bit-Prozessoren ergibt sich auch noch dadurch, daß die Register 16 Bit Informationen aufnehmen können und damit eine volle Adresse speichern können.

In Bild 6.11 sind einige Beispiele für den Befehlsaufbau des *9900* gezeigt. Die Datentransport- und die Datenverarbeitungs-Befehle, bei denen nur Register beteiligt sind, benötigen nur ein Befehlswort. Ebenso genügt ein Befehlswort, wenn im Befehl ein oder zwei Speicherplätze indirekt über Register als Zeiger adressiert werden. Zwei-Wort-Befehle werden für das unmittelbare Laden eines Registers verwendet. Wie üblich, enthält das zweite Befehlswort das zu ladende Bitmuster.

Zwei-Wort-Befehle kommen auch dann zur Anwendung, wenn ein Operand in einem Register steht und der zweite durch absolute Adressierung, beim *9900* heißt dies *direkte* oder *symbolische Adressierung,* bestimmt ist. Bei absoluter Adressierung muß das betreffende Quell- bzw. Zielregisterfeld den Inhalt **0000** haben. Werden beide Operanden absolut adressiert, so muß das zweite Befehlswort die Adresse der Datenquelle und das dritte Befehlswort die Adresse des Datenziels bzw. des Ergebnisses enthalten.

Bei Index-Adressierung wird im Operandenfeld der gleiche Code **10** für den Adreßtyp wie bei absoluter Adressierung verwendet. Der Prozessor erkennt einen Befehl mit Indexadressierung daran, daß in dem dazugehörigen Registerfeld eines der Register 1 bis 15 spezifiziert ist, während bei absoluter Adressierung dort nur **0000** stehen darf.

Außer den sonst üblichen arithmetischen Befehlen für Addition, Subtraktion, Decrementieren und Incrementieren kennt der *9900* auch jeweils einen Befehl für das Multiplizieren und Dividieren. Beim Multiplikations-Befehl werden zwei 16-Bit-Binärzahlen zu einer 32-Bit-Zahl multipliziert. Beim Dividieren erfolgt die Division einer 32-Bit-Zahl durch eine 16-Bit-Zahl.

Die Befehle für logische Operationen und für Vergleiche entsprechen im wesentlichen den Befehlen anderer Mikroprozessoren. Für die UND- bzw. ODER-Verknüpfungen kann eine unmittelbar adressierte Maske, für die EXKLUSIV-ODER-Verbindung aber nur der Inhalt eines Registers verwendet werden.

Die Ergebnisse von Rechnungen, logischen Operationen und Vergleichen werden in üblicher Weise durch Setzen oder Löschen einzelner Bit im Programm-Status-Register angezeigt. Dieses Register (Bild 6.12) enthält in den Bit Nr. 12 bis 15 auch noch eine Interrupt-Maske, durch die die Priorität von bis zu 16 verschiedenen Ebenen für Programm-Unterbrechungen festgelegt werden kann.

Auch die Sprungbefehle entsprechen im wesentlichen den Befehlen

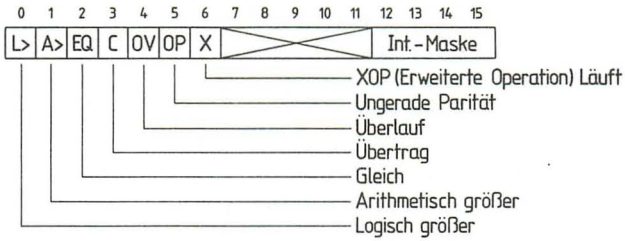


Bild 6.12 Aufbau des Programm-Status-Registers beim 9900

anderer Mikroprozessoren. Bei bedingten Sprüngen ist auch relatives Springen mit einer Sprungweite zwischen −128 und +127 Worten möglich. Die Verschiebung wird in den Bit Nr. 8 bis 15 im ersten Befehlswort als Zweier-Komplementzahl angegeben und, durch den Prozessor mit zwei multipliziert, zum Stand des Befehlsadreß-Registers hinzuaddiert oder von diesem abgezogen.

Im Gegensatz zu anderen Prozessoren werden unbedingte Sprungbefehle, bei denen die Sprungweite über den ganzen Adreßraum gehen kann, mit **BRANCH** bezeichnet. Bei diesen Befehlen kann die Sprungziel-Adresse nach den allgemeinen Regeln der Adressierung mit 5 verschiedenen Adressierarten sowohl in einem Register als auch in einer Speicherstelle stehen. Ähnlich aufgebaut sind auch die Einsprungbefehle in ein Unterprogramm, bei denen zusätzlich noch der Stand des Befehlsadreß-Registers und gegebenenfalls des Arbeitsregister-Zeigers gerettet wird.

Eine weitere Besonderheit des *9900* ist seine programmierbare Ein-/Ausgabe-CRU (COMMUNICATION REGISTER UNIT). Mit Hilfe der hierfür zuständigen Befehle ist es möglich, einzelne Bit in beliebigen Speicherstellen zu prüfen, zu setzen oder zu löschen. Da der *9900* keine besonderen Ein- und Ausgabebefehle kennt, können alle Port beliebig im Speicherraum angeordnet werden. Sie können also mit allen eine Speicherstelle betreffenden Befehlen angesprochen werden. Die dadurch mögliche Einzelbit-Verarbeitung ist besonders für alle Aufgaben in der Steuerungstechnik sehr nützlich.

Aus dem bisher Gesagten ist zu erkennen, daß der *9900* ein sehr vielseitiger Prozessor ist, der dem Programmierer sehr viele Möglichkeiten für einen wirkungsvollen Programmaufbau gibt. Obwohl der Befehlssatz

logisch aufgebaut ist, dürfte es aber recht schwer sein, diesen Prozessor noch allein in hexadezimaler Maschinensprache zu programmieren. Wenn man sich dagegen von vornherein nur der Assemblersprache bedient, ist die Programmierung auch nicht schwieriger als bei anderen Prozessoren. Man muß allerdings einige Mühe aufwenden, um die sehr großen Möglichkeiten dieses Prozessors auch auszunutzen, kann dann aber auch sehr komplexe Programmieraufgaben beherrschen.

7
Hinweise für die praktische Programmierarbeit

Ein Mikroprozessor-Programm kann man auf verschiedene Weise erstellen. Man kann

- die Befehlsfolge niederschreiben und dann hexadezimal in Maschinensprache kodieren
- mit einem Mikroprozessor-Entwicklungsgerät sofort in der Assemblersprache programmieren, die durch das Gerät dann in Maschinensprache übersetzt wird,
- das Programm in einer höheren Programmiersprache schreiben und es dann mit Hilfe eines Compiler-Programms auf einem Computer übersetzen lassen.

Ein Anfänger sollte aber nicht versuchen, nur mit der Assemblersprache zu arbeiten und alles andere einem Computer überlassen oder sogar gleich mit einer höheren Programmiersprache beginnen. Die Kenntnis der Maschinensprache ist für alle Methoden der Programmierarbeit nicht nur von Nutzen, sondern eigentlich Voraussetzung, wenn man zu effektiven, gut laufenden Programmen kommen will.

Da dieses Buch in erster Linie für Anfänger bestimmt ist, befaßt es sich in diesem Kapitel auch vorwiegend mit der zuerst genannten Art der Programmierung, wenn diese auch für die spätere praktische Arbeit keine große Bedeutung hat, denn für alle etwas längeren Programme, die also mehr als ca. 300 bis maximal 500 Befehle haben, ist eine gewisse Hilfe durch einen Computer fast unerläßlich.

Für die kommerzielle Programmerstellung wird man sich darum im allgemeinen eines *Mikroprozessor-Entwicklungsgerätes* bedienen. Diese Entwicklungscomputer gibt es inzwischen in vielen Ausführungen in der

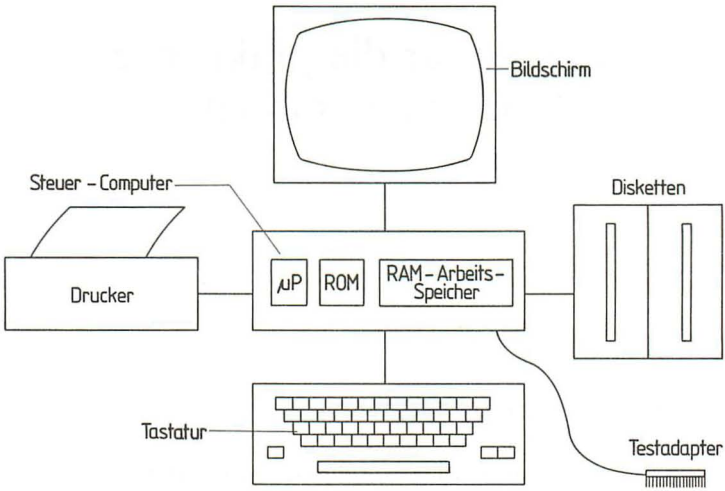


Bild 7.1 Schema eines Mikroprozessor-Programmentwicklungsgeräts

Preisgrößenordnung zwischen 10 000 und 50 000 DM, und es bedarf sorgfältiger Überlegungen, welches Gerät für die vorliegenden Aufgabenstellungen am geeignetsten ist (Bild 7.1).

Geräte, die noch mit dem Lochstreifen als Zwischenspeicher arbeiten, sind heute überholt und nicht zu empfehlen. *Kassettenspeicher* sind nur für sehr billige Geräte akzeptabel. Im allgemeinen werden *Diskettenspeicher* verwendet, die einen schnellen Zugriff zu allen schon früher entwickelten Programmen und Programmteilen gestatten.

Eine *Eingabetastatur* mit vollem Alphabet und ein alphanumerischer *Drucker* mit ausreichender Druckbreite sollte immer vorhanden sein. Im Prinzip genügt für das Drucken auch eine Fernschreibmaschine, jedoch ist heute der Preisunterschied zu einem leistungsfähigeren Nadeldrucker nicht mehr so groß, so daß letzterer vorzuziehen ist.

Sehr nützlich ist auch ein *Bildschirmgerät (englisch CRT = CATHODE RAY TERMINAL).* Auch hier gibt es heute schon recht preiswerte Terminals, bei denen Bildschirm und Tastatur zusammengefaßt sind.

Der eigentliche Computer muß genügend große *RAM-Kapazität* besitzen (mindestens 16 K, besser noch 32 K), die als *Arbeitsspeicher* verwendet wird.

Ebenso wichtig wie die Hardware ist die *System-Software,* die zum Gerät lieferbar ist. Man braucht zumindest ein voll ausgebautes Assemblier-Programm, ein Programm zum Einlesen, Ändern, Verschieben und Verbinden von Teilen des Anwenderprogramms und eine Fehlersuch-(DEBUGGING-)Routine, die auch den Ersatz des Prozessors in der Schaltung durch den Prozessor des Entwicklungsgerätes mit Hilfe eines *Testadapters* erlauben sollte. Die Namen, mit denen die verschiedenen Hersteller die einzelnen Hilfsprogramme für die Programmentwicklung anbieten, sind leider sehr unterschiedlich und darum verwirrend.

Man muß sich darum schon der Mühe unterziehen, die Funktionen der einzelnen Geräte genau zu studieren und miteinander zu vergleichen. In diesem Buch kann hierauf nicht im einzelnen eingegangen werden.

7.1 Programmieren lernen

Wie bei jeder etwas anspruchsvolleren Tätigkeit, so kann man auch beim Programmieren nur dann Erfolg haben, wenn man sowohl das rein Handwerkliche beherrscht als auch durch entsprechende Erfahrung weiß, wie gewisse Probleme anzugehen sind. Die handwerklichen Grundkenntnisse, d.h. also die Beherrschung des Befehlssatzes, muß man sich erarbeiten. Das Programmieren selbst lernt man aber nur durch eigene Erfahrung, wozu durchaus auch das Studium und die Analyse schon vorhandener Programme gehören kann.

Das Besondere bei der Mikroprozessor-Technik liegt nun darin, daß, im Gegensatz zu den Gepflogenheiten in der Großcomputertechnik, die Tätigkeit des eigentlichen Entwicklers, also desjenigen, der ein Mikroprozessor-System in seinem Gerät verwenden will, nicht ganz von der Tätigkeit des Programmierers zu trennen ist. Der Geräteentwickler braucht nicht die Erfahrung des Programmierers, er sollte aber zumindest das Handwerkliche der Programmiertechnik soweit beherrschen, daß er erkennen kann, welche Möglichkeiten ein Mikrocomputer bietet und wie diese im Prinzip zu verwirklichen sind. Mechanik und Elektronik lassen sich bei modernen Geräten nicht mehr so einfach trennen, daß sie durch zwei getrennte Entwicklungsteams bearbeitet werden können. Ein bezüglich Aufwand und Leistung wirklich optimales Gerät kann nur entstehen, wenn auch der Mechanikentwickler genügende Kenntnisse

über den im Gerät zu verwendenden Mikroprozessor und seine Programmierung besitzt.

7.1.1 Besuch von Lehrgängen und Seminaren

Mikroprozessor-Lehrgänge und Seminare werden sowohl von den Herstellern als auch von unabhängigen Instituten veranstaltet. Unterscheiden muß man hierbei reine Einführungsseminare, die meist nur einen mehr oder minder konzentrierten Überblick über das Gesamtgebiet der Mikroprozessor-Technik geben, und echte Lehrgänge, bei denen die Beherrschung der Technik geübt wird.

Einführungsseminare kann man vielleicht noch ohne besondere Vorbereitung besuchen. Der Vortragende wird dort berücksichtigen, daß er zu Nicht-Fachleuten spricht, und sich dementsprechend im Stoff beschränken. Man kann dort nur eine Fülle von Anregungen für weiteres Nachdenken erwarten. Bei Lehrgängen, oder, wie man heute auch oft sagt, bei *Workshop-Seminaren,* wird echte Mitarbeit verlangt, und da der Stoff oft sehr konzentriert angeboten wird, muß man dort meist schon recht solide Grundkenntnisse mitbringen. Intensives Selbststudium mit späterer Vertiefung der Kenntnisse durch einen Lehrgang ist im allgemeinen der umgekehrten Reihenfolge vorzuziehen.

7.1.2 Selbststudium mit Mikroprozessor-Lerngeräten

Es gibt heute eine ganze Reihe von *Mikroprozessor-Lern- und Übungsgeräten* (Bild 7.2), die für das Selbststudium gut geeignet sind. Man versteht hierunter Geräte, die neben einem kompletten Mikrocomputer noch eine Tastatur und eine Anzeige besitzen, mit deren Hilfe ein in hexadezimaler Maschinensprache geschriebenes Programm in den RAM-Speicher des Computers eingegeben werden kann. Damit dies möglich ist, muß der Lerncomputer in einem ROM gespeichert ein sogenanntes *Monitorprogramm* besitzen, das die Eingabe, Änderung und Prüfung des vom Anwender geschriebenen Programms erlaubt. Die an ein derartiges Lernsystem zu stellenden Anforderungen sowie Vor- und Nachteile einzelner Ausführungsformen dieser Geräte werden im nächsten Abschnitt besprochen.

Die Preise von Mikroprozessor-Lern- und Übungsgeräten liegen, wenn sie komplett ausgestattet sind, in der Größenordnung von 1000 bis

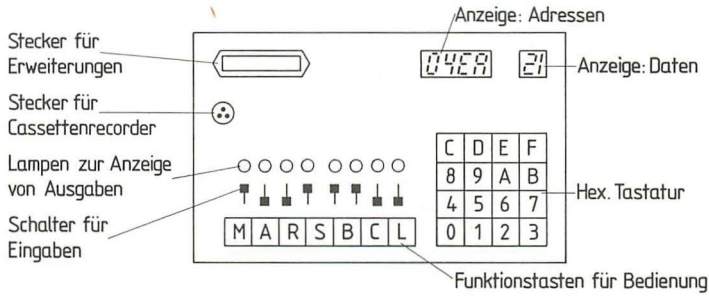


Bild 7.2 Schema eines Mikroprozessor-Lern- und Übungsgeräts

1500 DM. Neben dem Gerät braucht man aber unbedingt noch ein entsprechendes Lehrbuch, und da wird es schon etwas schwierig. Bei einigen Geräten stehen die Lehrbücher nur in englischer Sprache zur Verfügung, bei anderen ist der Originaltext ziemlich phantasielos ins Deutsche übersetzt und darum auch nicht so ohne weiteres verständlich. Man sollte darum für das Selbststudium ein Gerät wählen, zu dem es ein in verständlicher Sprache geschriebenes deutsches Lehrbuch gibt, wobei es dabei ganz bestimmt nicht auf die Zahl der Seiten, die geliefert werden, sondern auf den Inhalt ankommt. Darüber hinaus sollte man sich das Originalhandbuch des Herstellers beschaffen, dessen Mikroprozessor in dem Gerät eingebaut ist. Man wird sicherlich auch noch einige weitere Lehrbücher kaufen, die die Problematik der Programmierung einführend behandeln, wozu auch dieses Buch gehört.

Für den Anfang ist es nicht so besonders wichtig, mit welchem Prozessortyp man beginnt, in die Programmiertechnik einzudringen. Es ist gewiß nützlich, ein Lern- und Übungsgerät zu benutzen, das den gleichen Prozessor eingebaut hat, den man vielleicht später in der Praxis benutzen will. Unbedingt notwendig ist dies nicht, denn wenn man erst einmal grundsätzlich mit den Programmiermethoden für einen Typ vertraut ist, bedeutet es meist keine sehr große Schwierigkeit, auch auf einen anderen Typ überzugehen.

Man sollte von vornherein versuchen, sich selbst in der Praxis mögliche Aufgaben zu stellen und diese zu lösen. Es ist nicht besonders sinnvoll, nur die in manchen Lehrbüchern beschriebenen Primitivprogramme, bei denen immer nur ganz wenige Befehle auszuführen sind, zu

lange zu üben. Gerade die Kombination verschiedener Befehle, die für ein wirkliches Programm notwendig sind, bereitet am Anfang die meisten Schwierigkeiten und sollte darum auch von Anfang an geübt werden.

Das Arbeiten mit diesen kleinen Lerngeräten kann zum faszinierenden Hobby werden. Es ist ein echtes Erfolgserlebnis, wenn zum erstenmal eine vollständige Aufgabe durch ein selbstentwickeltes Programm gelöst wird. Wer sich einmal in dieser Weise etwas näher mit der Mikroprozessor-Technik befaßt, kann daraus ein Hobby entwickeln, das ihn über die Tätigkeit als Freizeitbeschäftigung hinaus echte berufliche Chancen bietet.

7.2 Anforderungen an Mikroprozessor-Lern- und Übungsgeräte

Die wesentlichen Eigenschaften einer Reihe von Lerncomputern sind in einer Tabelle im Anhang zusammengestellt. Obwohl die Geräte in ihrem grundsätzlichen Aufbau recht ähnlich sind und alle die gestellte Aufgabe, Programmentwicklung in hexadezimaler Maschinensprache zu ermöglichen, erfüllen, unterscheiden sie sich doch in vielen Einzelheiten sowohl beim Aufbau als auch beim Monitor-Programm.

7.2.1 Hardware-Aufbau

Bezüglich des RAM-Speicherbereichs bestehen für ein reines Lern- und Übungsgerät keine besonders hohen Anforderungen. Ein Anfänger wird kaum in der Lage sein, ein Programm zu schreiben, das mehr als hundert Befehls-Byte Speicherraum benötigt. Und so dürfte wohl die bei vielen Geräten vorgesehene Mindestbestückung mit 1/4 KByte RAM-Speicherraum im allgemeinen ausreichen. Bei fast allen Geräten ist auch noch eine Erweiterungsmöglichkeit vorgesehen, sehr oft allerdings auf einer Zusatzplatine.

Für eine bequeme Bedienung außerordentlich wichtig ist die Ausführung des Tastenfeldes. Die Tasten müssen gut bedienbar sein und sollen einen kleinen, deutlich fühlbaren Weg machen, wenn sie gedrückt werden. Reine Berührungstasten sind ungeeignet. Auch Tasten mit Mehr-

fachfunktionen sind schwieriger zu bedienen als Einfachfunktionstasten. Gerade bei einem Lerngerät kommt es sehr darauf an, daß nicht noch durch Bedienungsfehler zusätzliche Fehlermöglichkeiten zu den unvermeidlichen Programmierfehlern hinzukommen.

Die Anzeige sollte mindestens 6, besser aber noch 8 Stellen haben, damit jeweils eine Adresse und deren Inhalt oder die Inhalte mehrerer Register zusammen sichtbar gemacht werden können.

Sehr günstig ist es, wenn das Gerät schon einige Leuchtdioden und Schalter besitzt, über die Ein- und Ausgangsoperationen simuliert werden können. Ist dies nicht der Fall, sollte zumindest ein geeigneter Stekkerausgang vorhanden sein, an die man dann selbst eine kleine Zusatzplatine zur Darstellung von Ein- und Ausgaben über einen Port anschließen muß.

Jeder Mikroprozessor braucht auch eine Stromversorgung. Die neueren Prozessortypen begnügen sich meist mit +5 Volt, und so sollte der Lerncomputer möglichst auch so aufgebaut werden, daß eine einfache Stromversorgung möglich ist. Besonders leicht zu handhaben sind die Geräte, die zusammen mit der Stromversorgung in ein Gehäuse eingebaut sind, so daß für die Betriebsbereitschaft lediglich die Verbindung mit einer Steckdose hergestellt zu werden braucht.

Einige Lerngeräte bestehen nur aus einer großen Platine, während andere als Mehrplatinengeräte mit entsprechenden Steckverbindungen aufgebaut sind. Alle Geräte haben einen oder mehrere Stecker, über die Zusatzplatinen oder andere Zusatzgeräte angeschlossen werden können, wodurch manchmal eine Erweiterungsmöglichkeit bis zum einfachen Mikroprozessor-Entwicklungsgerät besteht.

Andere Geräte wiederum sind so aufgebaut, daß entweder das ganze Gerät oder Teile davon für selbst zu entwickelnde Mikroprozessor-Steuerungen Verwendung finden können. Weder das eine noch das andere sollte das Gerät jedoch so verfälschen, daß der Hauptzweck eines Lerngerätes, nämlich einfache Einführung in die Mikroprozessor-Programmierung, verlorengeht.

7.2.2 Anforderungen an das Monitorprogramm

Im allgemeinen sind Geräte, die mit einem echten Monitorprogramm und hexadezimaler Eingabe aller Befehle arbeiten, leichter zu bedienen als andere Geräte, die einen sogenannten *Zeilen-Assembler* (LINE BY

LINE ASSEMBLER) verwenden. Unter letzterem versteht man ein Programm, das Befehle, die als mnemonische Kürzel eingegeben werden, Zeile für Zeile in Maschinencode umwandelt. Man muß bei diesen Geräten also das Programm alphanumerisch eingeben und braucht auch eine entsprechende Tastatur. Da bei diesen Tastaturen jede der Tasten immer zwei oder mehrere Funktionen hat, sind schon Fehler bei der Eingabe recht häufig.

Das Monitorprogramm sollte also vor allem zunächst eine einfache irrtumsfreie Eingabe des selbstentwickelten Programms in den Programmspeicher des Geräts ermöglichen. Hierzu ist eine automatische Adressenfortschaltung, die nach Eingabe jeweils eines Programms Byte zur nächsten Speicheradresse schaltet, sehr nützlich. Das Monitorprogramm sollte auch so aufgebaut sein, daß man jede Eingabe, und zwar sowohl Adresse als auch Inhalt, in der Anzeige erkennen kann. Dabei muß es möglich sein, einen Eingabefehler zu korrigieren, ohne daß die gesamte Programmeingabe wiederholt zu werden braucht. Es muß weiterhin die Möglichkeit bestehen, alle eingegebenen Daten zur Kontrolle auf die Anzeige zu bringen und dabei gegebenenfalls zu ändern.

Mikroprozessor-Lerngeräte dienen aber nicht nur der Programmentwicklung, sondern auf ihnen soll ein vom Anwender geschriebenes Programm auch laufen, um nachzuweisen, daß es seinen Zweck erfüllt. Durch entsprechende Befehle des Monitorprogramms muß es also möglich sein, ein Anwenderprogramm von einer beliebigen Adresse beginnend zu starten und das Ergebnis in der Anzeige zu zeigen oder über ein Ausgabe-Port auszugeben.

Besonders wichtig für ein Lerngerät sind Hilfsmittel im Monitorprogramm, um ein selbstgeschriebenes Programm zu testen und gegebenenfalls darin befindliche Fehler auszumerzen. Ein Lerngerät soll darum auch dem *Programmlauf Schritt für Schritt* gestatten, d.h., es wird immer nur ein Befehl abgearbeitet, wonach die Möglichkeit besteht, Speicher- und Registerinhalte zu kontrollieren. Ebenso sollte bei jedem Lerngerät das Setzen und Löschen von *Haltpunkten* (BREAK POINTS) möglich sein. Hierunter versteht man besondere Befehle, durch die ein laufendes Programm an einer ganz bestimmten Adresse angehalten wird. Man kann dann untersuchen, wie der Inhalt der einzelnen Register nach dem Anhalten des Programms ist, um daraus zu erkennen, ob der Programmablauf den Wünschen des Programmierers entsprochen hat.

Eine kleine, aber sehr nützliche Erleichterung für den Programmtest

ist es, wenn bei der Abfrage der Registerinhalte das Programm-Status-Wort nicht hexadezimal, sondern binär angezeigt wird. Man kann dann sehr viel schneller erkennen, ob ein falscher Programmlauf, etwa durch Nichtbeachtung des Carry-Bit, erfolgt ist.

Da der Arbeitsspeicher der Mikroprozessor-Lerngeräte im allgemeinen nur klein ist und im übrigen seinen Inhalt verliert, wenn die Stromversorgung abgeschaltet wird, muß die Möglichkeit bestehen, entwickelte Programme auf andere Weise dauernd zu speichern. Im allgemeinen sind die Geräte darum so gebaut, daß handelsübliche, billige Kassettenrekorder angeschlossen werden können, die dann als Programmspeicher dienen. Auch die Bedienung des Kassettenrekorders erfolgt dann durch einen Teil des Monitorprogramms, wobei es nützlich ist, wenn man jedem abgespeicherten Programmteil eine *Kennummer* (FILE NUMBER) geben kann, um es so leichter wiederzufinden und in den Arbeitsspeicher für einen Programmlauf laden zu können.

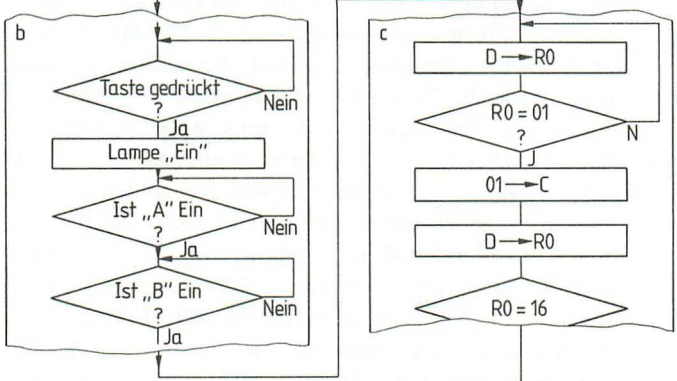
Insgesamt gesehen sollte der Monitor nicht zu kompliziert aufgebaut sein, denn sonst kann der Anfänger mehr Fehler durch falsche Bedienung des Gerätes als beim Programmieren machen. Erwünscht ist es aber, daß Teile des Monitorprogramms, zum Beispiel die Anzeige und die Tastaturanfrage oder gewisse Umcodierungen, als Unterprogramme zur Benutzung in einem Anwenderprogramm abgerufen werden können. Was in dieser Hinsicht möglich ist und wie es gemacht wird, muß genau in der Betriebsanleitung beschrieben sein. Diese Anleitung sollte auch den vollständigen Ausdruck des Monitorprogramms enthalten. Das Studium dieses Programms ist meist sehr nützlich, kann man doch viele Erkenntnisse, wie Einzelaufgaben zu lösen sind, durch das Studium gut geschriebener Programme erlernen.

7.3 Die Schritte der Programmentwicklung

Bei der Lösung einer Aufgabe mit Hilfe eines Mikroprozessorprogramms soll man grundsätzlich schrittweise vorgehen und den nächsten Schritt immer erst dann beginnen, wenn der vorherige klar abgeschlossen ist (Bild 7.3). Dies trifft natürlich in erster Linie für etwas größere Programme zu, aber der Anfänger tut gut, sich auch bei kleinen Programmen an die Regeln, die im folgenden beschrieben werden, zu halten.

a

Beim Betätigen einer Drucktaste soll eine Kontrollampe anzeigen, daß der Startbefehl gegeben ist, der Motor soll nur anlaufen, wenn 1. die Stillstandssicherung einmal angesprochen hat, 2. Papier richtig eingelegt ist

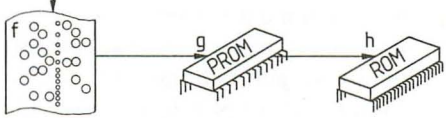


d

```
TASTE   REDD, RO O          TASTE ABFRAGEN
        TMI,  RO H'01'
        BCFR, NP TASTE
        LODI, RO H'10'
        WRTC, RO            Kontrollampe ein
```

e

```
2214 1113 70   TASTE LODI, RO O      TASTE ABF.
2215 1114 B001       TMI,  RO H'01'
2216 1116 987B       BCFR, NP        TASTE
2217 1118 0410       LODI, RO H'10'
2218 111A C3         WRTC, RO
```



7.3.1 Aufgabenbeschreibung

Jede Programmierarbeit beginnt mit einer detaillierten Beschreibung der zu lösenden Aufgabe. Die Beschreibung kann zunächst allein mit Worten erfolgen, aber durchaus auch von einem Blockschaltbild oder von einem Schaltplan begleitet sein. Es kommt darauf an, daß hier wirklich alle Forderungen, die durch das Programm erfüllt werden sollen, enthalten sind, wobei man zunächst noch keinerlei Rücksicht auf den möglichen Lösungsweg zu nehmen braucht.

Unbedingt notwendig ist es jedoch, daß in der Aufgabenbeschreibung alle Maßnahmen für die Bedienung genau beschrieben werden. Es soll also möglichst genau festgelegt werden, welche Bedienungselemente benötigt werden und in welcher Weise sie zu bedienen sind. Denn gerade die Festlegung der Bedienung eines Geräts ist Aufgabe für den Gerätekonstrukteur und sollte nicht dem Programmierer überlassen bleiben.

Auf der anderen Seite muß die Beschreibung auch genaue Angaben darüber enthalten, was und auf welche Weise der Mikroprozessor steuern soll bzw. welche Ausgangsinformationen vom Mikroprozessor erwartet werden. Ein Mikroprozessor selbst ist nicht in der Lage, irgendwelche Leistung zu steuern, so daß hierfür andere elektrische oder elektronische Elemente nachgeschaltet sein müssen.

Man muß von vornherein festlegen, welche Steuersignale der Mikroprozessor abgeben soll, damit die Schnittstelle zur Leistungssteuerung genau definiert ist.

In gleicher Weise muß natürlich auch eine Schnittstelle definiert werden, wenn irgendwelche Daten mit anderen Geräten ausgetauscht wer-

Bild 7.3 Schritte bei der Entwicklung eines Mikroprozessor-Programms
Die Entwicklung eines Programms für ein Mikroprozessor-System erfolgt zweckmäßigerweise in mehreren Schritten nacheinander.
Zunächst muß die zu lösende Aufgabe so klar wie möglich in Worten beschrieben werden (a). Daraus wird ein erstes Flußdiagramm, auch Ablaufdiagramm genannt, entwickelt (b). In diesem wird der zu steuernde Vorgang in eine Folge von Schritten zerlegt, wobei aber noch keine Rücksicht auf den Befehlssatz des verwendeten Prozessors genommen wird. Als nächstes wird das Befehlsdiagramm aufgestellt (c), das den Vorgang so darstellt, wie er durch Befehle gesteuert werden soll.
Es folgt die erste Niederschrift der Befehle in der symbolischen Assemblersprache (ein Vorgang, der auch Editieren genannt wird). In µP-Entwicklungssystem entsteht dann durch Assemblieren mit ausgerechneten Adressen und richtig codierten Befehlen (e) das Programm, das zunächst auf einer Platte oder einem Lochstreifen zwischengespeichert (f) und dann elektrisch in ein PROM eingeschrieben (g) wird.

den sollen. Hier wird man im allgemeinen nicht umhinkommen, ein einfaches Blockschema zu entwerfen, in dem der Mikrocomputer nur ein Kästchen mit diversen Ein- und Ausgabeleitungen darstellt.

In jedem Fall ist es nötig, bereits in dieser ersten Entwurfsphase weitgehende Übereinstimmung zwischen allen an der Entwicklung Beteiligten herzustellen. Die eigentliche Programmierarbeit sollte erst dann beginnen, wenn die allgemeine Aufgabenbeschreibung, also das Pflichtenheft, von allen Beteiligten anerkannt und damit zur Grundlage für die Entwicklungsarbeit gemacht worden ist.

7.3.2 Festlegung der Hardware

Wenn eine Aufgabenbeschreibung erstellt ist, ist es als nächstes notwendig, abzuschätzen, welche Bauelemente notwendig sind, um ein vollständiges Mikroprozessor-System für die gestellte Aufgabe aufzubauen. Bezüglich des Prozessortyps wird man sich für den Typ entscheiden, mit dem man schon die meisten Erfahrungen gemacht hat. Abschätzen muß man jedoch den Bedarf an Speicherplätzen sowohl für das Programm als auch für feste und variable Daten sowie die Anzahl der notwendigen Ein- und Ausgabeleitungen in das System. Soweit nicht Preisgründe dagegen sprechen, wird man die von den Mikroprozessor-Herstellern angebotenen und sehr vielseitig verwendbaren Ein- und Ausgabebausteine vorsehen, die die Programmierarbeit erleichtern können.

Gerade bei den Ein- und Ausgabeleitungen sollte man von vornherein klare Verhältnisse schaffen, denn jeder dieser Bausteine, die meist im Preis etwa in gleicher Größenordnung liegen wie ein kompletter Mikroprozessor, hat nur eine bestimmte Anzahl von Port, so daß man sich schon von vornherein genau überlegen muß, wieviel Leitungen man wirklich braucht, um nicht wegen einer zusätzlich notwendigen Leitung einen weiteren Baustein verwenden zu müssen.

Insbesondere muß man auf die Beschränkung der zur Verfügung stehenden Ein- und Ausgabeleitungen und gegebenenfalls auch des Speicherraums Rücksicht nehmen, wenn man zur Erleichterung der Hardware-Entwicklung mit fertigbezogenen Computer-Platinen arbeitet, was für den Anfang durchaus empfohlen werden kann.

Die fertigen Mikrocomputer-Platinen, die heute von verschiedenen Herstellern im Europa-Format 100 mm × 160 mm angeboten werden, und die meist auf der Platine neben dem Prozessor noch eine ganze Anzahl

von Speicherelementen und Ein- und Ausgabeports enthalten, stellen für die meisten Aufgaben, mit denen ein Anfänger sich zunächst beschäftigt, ausreichend Kapazität zur Verfügung. Es kann aber doch vorkommen, daß man irgendwie nicht auskommt. Dann sollte in Erwägung gezogen werden, das Pflichtenheft etwas zu ändern, ehe man beginnt, noch zusätzliche Bauelemente vorzusehen.

7.3.3 Beschreibendes Flußdiagramm (Algorithmus)

Ein Mikrocomputer kann nur einen Befehl nach dem anderen bearbeiten. Er arbeitet rein sequentiell, was ein gewisses Umdenken für alle Elektronik-Entwickler erfordert, denn in einer normalen Schaltung wird mit vielen parallel geführten Strompfaden gearbeitet, durch die gleichzeitig Strom fließt.

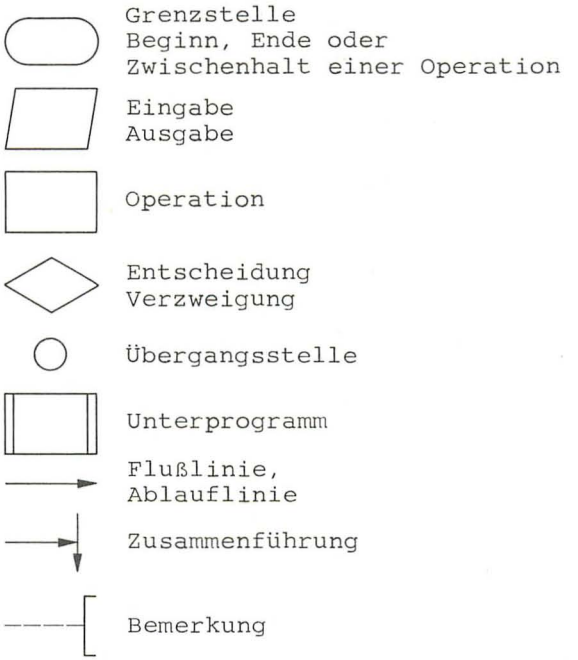


Bild 7.4 Symbole für einen Programmablaufplan (Flußdiagramm)

Das beste Mittel, einen sequentiellen Programmablauf darzustellen, ist das Flußdiagramm, dessen wenige Symbole man schnell erlernen kann (Bild 7.4). In Form eines Flußdiagramms, als allgemeiner Lösungsweg auch *Algorithmus* genannt, kann man die vom Mikrocomputer geforderte Arbeit in Teilabschnitten darstellen. Man wird zunächst mit groben Aussagen arbeiten, und insbesondere die während des Programmes zu treffenden Entscheidungen herausstellen und dann das Programm immer weiter verfeinern. Dabei wird man schon weitgehend an die Unterprogrammtechnik denken und kann am Anfang einzelne Programmabschnitte einfach als Unterprogramm bezeichnen, das dann zu einem späteren Zeitpunkt genauer definiert wird.

In diesem Stadium der Programmentwicklung sollte man sich noch keine Gedanken darüber machen, wie das Programm dann tatsächlich mit Hilfe eines Mikroprozessors verwirklicht wird. In dem ersten beschreibenden Flußdiagramm braucht also überhaupt noch nicht auf die Befehls-Struktur des verwendeten Prozessors Rücksicht genommen zu werden. Es sollen weder bestimmte Register noch bestimmte Speicherstellen benannt werden, sondern es geht im wesentlichen darum, die verschiedenen Teilaufgaben, die im Programm zu erfüllen sind, und die dafür eventuell notwendigen Entscheidungskriterien darzustellen. Auch in diesem Stadium der Programmentwicklung werden sicherlich noch Abstimmungen mit der mechanischen Entwicklung notwendig sein, aber Anpassungen und Änderungen nur auf dem Papier sind verhältnismäßig leicht und einfach möglich.

Schon hier eine kleine Empfehlung: Benutzen Sie bei der Programmierarbeit keine Tinte und keinen Kugelschreiber, sondern einen Bleistift mit dem dazugehörigen Radiergummi, um Änderungen und Umstellungen einfach durchführen zu können. Es ist auch sinnvoll, einen verhältnismäßig großen Papierbogen zu benutzen und zunächst die einzelnen Kästchen und Rhomben mit weitem Abstand zu zeichnen, um später noch etwas einfügen zu können.

7.3.4 Befehlsdiagramm und Programmentwurf

Wenn jetzt das beschreibende Flußdiagramm, das den Ablauf des zu schreibenden Diagramms im wesentlichen festlegt, vorliegt, muß daraus das eigentliche Programm geschaffen werden. Je nach Übung und Erfahrung des Programmierers kann dieser nun zunächst Teile des Flußdia-

gramms in ein echtes Befehlsdiagramm umsetzen, indem also genau festgelegt wird, welche Register Verwendung finden und wie der Datentransport im Computersystem im einzelnen vonstatten geht. Oder er kann bereits beginnen, Teile des gesamten Programms in der Assembler-Sprache zu formulieren, wenn der Programmablauf leicht zu überblikken ist. In jedem Fall muß man jedoch vorher eine Liste anlegen, aus der die Adressen für benötigte Datenspeicherplätze und Ein- und Ausgangskanäle verzeichnet sind. Für diese Adressen wird man symbolische Ausdrücke wählen, die dann später in das Programm eingesetzt werden können. Für die Wahl der symbolischen Ausdrücke hat der Programmierer jede Freiheit. Es hat sich aber gezeigt, daß die insbesondere in amerikanischen Programmen sehr beliebten, oft unverständlichen Abkürzungen, die nur aus Konsonanten bestehen, für das Behalten nicht gerade besonders gut geeignet sind.

Gegebenenfalls sind willkürlich gewählte Namen oder Ausdrücke, die aus einem Buchstaben und einer Ziffer bestehen, besser handhabbar als Symbole, bei denen krampfhaft versucht wurde, eine Bedeutung hineinzulegen.

Die Liste der symbolischen Adressen sollte immer neben den eigentlichen Programmentwürfen zur Hand sein, denn man muß immer wieder darauf zurückgreifen. Oft ist es auch zweckmäßig, gewisse Konstanten oder sogar Variable zunächst nur durch einen symbolischen Ausdruck festzulegen und diese dann erst später mit ihrem wahren Wert in das Programm einzusetzen. In diesem Fall muß man auch für diese Werte eine entsprechende Liste führen, die immer auf dem laufenden sein muß.

Das Programm selbst kann man fast nie in einem Stück, vom ersten Befehl an beginnend, schreiben, sondern muß es in kleine Teilabschnitte aufteilen. Diese Teilabschnitte sollen in sich geschlossen sein, übersichtlich bleiben und in der Größenordnung etwa 50 Befehle nicht überschreiten. Im Extremfall kann jedes dieser Teilprogramme als Unterprogramm formuliert sein und dann durch das zusammenfassende Hauptprogramm nur angerufen werden.

Eine andere Möglichkeit der Programmierung stellt die Verwendung von MACROS dar. Hierbei handelt es sich um Programmabschnitte, die immer wieder in gleicher Weise benötigt werden. Wenn man diese einmal formuliert hat, können sie durch einen *Macro-Assembler,* also ein besonderes Programm auf einem Mikroprozessor-Entwicklungsgerät,

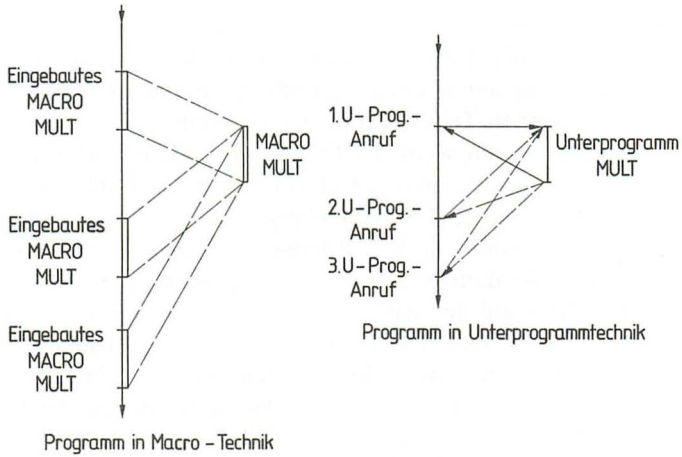


Bild 7.5 Der verschiedene Programmaufbau beim Einbau von MACROS und beim Einsprung in Unterprogramme
Mehrfach zu benutzende Programmteile können als Macro geschrieben und dann durch den Macro-Assembler überall dort in das Hauptprogramm eingebaut werden, wo sie benötigt werden.

Der gleiche Zweck kann auch mit der Unterprogrammtechnik erreicht werden.

selbsttätig in die verschiedenen Stellen des Hauptprogramms, wo sie benötigt werden, eingefügt werden.

Der wesentliche Unterschied zwischen einem Macro und einem Unterprogramm besteht also darin, daß das Unterprogramm in der Programmfolge nur einmal erscheint und beliebig oft durch spezielle Befehle angerufen werden kann. Ein Macro wird als Programm auch nur einmal geschrieben, dann aber in das Programm überall dort eingesetzt, wo es gebraucht wird. Im allgemeinen verwendet man die Unterprogrammtechnik, wenn etwas längere Programmabschnitte mehrfach Verwendung finden sollen und die Macro-Technik für kürzere Programmteile, die immer wieder vorkommen (Bild 7.5).

7.3.5 Codieren, Editieren, Assemblieren

Das zunächst auf Papier niedergeschriebene Programm muß nun so aufbereitet werden, daß es der Mikroprozessor versteht. Das heißt, es muß in die Maschinensprache des Prozessors übertragen werden. Diesen Vorgang bezeichnet man ganz allgemein als Codieren, er kann auf verschiedene Weise ausgeführt werden.

7.3.5.1 Arbeiten mit einem Lerngerät

Wenn als Hilfsmittel für die Programmierung nur ein Mikroprozessor-Lerngerät zur Verfügung steht, das die Eingabe der Befehle in hexadezimaler Codierung der Maschinensprache erfordert, muß die Assembler-Sprache zunächst mit Hilfe der Befehlstabellen in hexadezimale Maschinensprache umgewandelt werden. Dies ist eine Arbeit, die einige Sorgfalt und oft auch einiges Nachdenken erfordert, denn die Befehlstabellen, die von den Herstellern der Mikroprozessoren herausgegeben werden, sind nicht immer gerade sehr benutzerfreundlich aufgebaut. Da für den gleichen Befehl verschiedene Operationscodes in Frage kommen, je nachdem, welches Register daran beteiligt ist, geben manche Befehlstabellen überhaupt keine vollständige hexadezimale Codierung der Befehle an. Man muß also zunächst danach trachten, mit einer Befehlstabelle zu arbeiten, die wirklich für jede Befehlsvariante die vollständige hexadezimale Codierung zeigt, denn sonst ist das Codieren der Befehle viel zu mühsam. In diesem Sinne ist es eine sehr gute Übung, den Befehlssatz seines Mikroprozessors beherrschen zu lernen, wenn man selbst eine Befehlstabelle, mit allen notwendigen Angaben in genügend großer Schrift, übersichtlich aufgestellt, anfertigt und diese dann benutzt. Wenn man hierbei gezwungen ist, sich mit jedem einzelnen Befehl zu befassen, lernt man sehr viel, was der späteren Arbeit zugute kommt.

Zusammen mit der Codierung der Befehle muß jetzt auch die Adressenzuweisung erfolgen. Vorher ist ja bereits durch hardwaremäßigen Anschluß der entsprechenden Speicherelemente festgelegt, welche Adreßräume für das Programm und welche für die Daten bestimmt sind. Im allgemeinen wird es bei den ersten Programmteilen, die zu codieren sind, noch nicht möglich sein, den endgültigen Speicherraum, den dieser Teil des Programms später einnehmen soll, zu bestimmen. Man muß dann später die einzelnen Programmteile noch verschieben, was immer

eine gewisse Umrechnung erfordert, wenn nicht dafür auch wieder ein Computerprogramm verwendet wird.

Beginnt man das Programm zunächst für ein Mikroprozessor-Lerngerät zu schreiben, so muß man die Speicherplätze benutzen, die vom Hersteller des Gerätes für das Anwenden des Programms bestimmt sind. Man sollte aber nicht unbedingt mit dem allerersten möglichen Speicherplatz beginnen, denn es kann durchaus vorkommen, daß man im Laufe der Programmentwicklung merkt, daß zu Beginn des Programms irgend etwas vergessen wurde. Das Einfügen zusätzlicher Befehle ist einfacher, wenn man den Programmbeginn auch noch verschieben kann.

Bei der Adressenzuweisung für die einzelnen Befehle ist sorgfältig zu beachten, ob es sich um Ein-, Zwei- oder Drei-Byte-Befehle handelt. Wenn man das Ganze niederschreibt, sollte man wieder einen Bleistift benutzen und zwischen den einzelnen Befehlszeilen jeweils einige Zeilen Platz lassen, damit man später noch etwas einfügen kann, wenn beim Testen festgestellt wird, daß einige Befehle nicht richtig arbeiten.

Fehler passieren dem Anfänger oft bei der Adressenberechnung für Sprungbefehle. Am einfachsten ist es noch bei allen Sprungbefehlen mit absoluter Adressierung, bei denen rückwärts gesprungen wird, denn dort ist ja bereits die Sprungadresse bekannt und braucht nur richtig in den Befehl eingesetzt zu werden. Dies ist aber nicht der Fall bei Vorwärtssprüngen. Hier sollte man zunächst bei der Befehlscodierung anstelle der Befehls-Byte mit der Sprungadresse Klammern oder Punkte einsetzen, damit diese Byte bei der Adressenzählung nicht vergessen werden. Die eigentliche Sprungadresse kann man dann erst in den Befehl einsetzen, wenn man mit der Codierung bis zu dem Sprungziel gekommen ist und man dessen Adresse kennt. Genauso verfährt man zweckmäßigerweise auch bei relativen Sprüngen. Für die Ausrechnung der relativen Verschiebung bedient man sich einer Tabelle, wie sie im Anhang dieses Buches abgedruckt ist.

Auch sonst können beim Einsetzen der Adressen in die Befehle leicht Fehler vorkommen, wenn man nicht die für den betreffenden Prozessortyp gültigen Regeln bezüglich der Reihenfolge genau befolgt.

Ein in dieser Weise mit Hilfe einer Befehlstabelle codiertes Programm kann dann mit der Hexadezimal-Tastatur in das Lerngerät eingegeben werden. Hierbei sind auch wieder gewisse Vorschriften zu beachten, die man der Bedienungsanleitung dieses Gerätes entnehmen muß. Bevor

dann das Programm laufen kann, muß man noch zusätzlich die für das Programm benötigten Daten in die dafür vorgesehenen Speicherstellen eingeben. Dann beginnt das Testen, das im nächsten Abschnitt beschrieben wird.

7.3.5.2 Arbeiten mit einem Entwicklungs-Computer

Etwas anders ist das Vorgehen, wenn ein Programmentwicklungs-Computer mit alphanumerischer Tastatur zur Verfügung steht. Auch wenn man diese Geräte verwendet, muß man im allgemeinen einen Programmentwurf in Assemblersprache auf Papier schreiben, aber alle anderen Arbeiten werden dann durch den Computer erledigt. Die Geräte haben meist ein mit EDITOR bezeichnetes Betriebsprogramm, das der ersten Eingabe des Anwenderprogramms in das System dient. Hierdurch werden zunächst einmal die vom Programmierer für sein Programm ausgewählten Befehle computergerecht für die weitere Verarbeitung gespeichert, wobei gegebenenfalls schon grobe Fehler ausgemerzt werden können. Das zum erstenmal eingegebene Programm, das noch keine codierten Befehle und Adressen enthält, nennt man *Quellenprogramm* (SOURCE PROGRAM).

Die Eingabe des geschriebenen Programms erfolgt durch Eintasten eines Befehls nach dem anderen, wobei man ganz genau alle Regeln der betreffenden Assemblersprache beachten muß. Es kommt also auf jedes Komma und jeden Zwischenraum an. Offensichtlich falsche Eingaben, z.B. Buchstabenzusammenstellungen, die keinen sinnvollen Befehl ergeben, meldet der Computer, so daß sie korrigiert werden können. Wenn zur Tastatur auch noch eine Bildschirmanzeige oder eine sonstige alphanumerische Zeilenanzeige zur Verfügung steht, kann jede Eingabe sofort überprüft und ebenfalls verbessert werden.

Für die Eingabe der Adressen sowie aller sonstigen Zahlenwerte bestehen eine Reihe von Regeln, die man genau einhalten muß (siehe auch Bild 5.3). Für im Programm vorkommende Adressen kann man Symbole verwenden, auch für Daten sind diese zugelassen. Man muß jedoch dabei jeweils auch wieder die Regeln der entsprechenden Assemblersprache bezüglich maximaler Zeichenzahl für die Symbole beachten. Um die einzelnen Adressen, in denen Befehle und Daten gespeichert werden sollen, braucht man sich zunächst noch nicht zu kümmern. Die Festlegung der Adressen erfolgt vor dem nächsten Schritt der Assemblierung.

Vor Beginn mit dem eigentlichen *Assemblieren,* womit man die Umwandlung des in Assemblersprache geschriebenen Programms in ein lauffähiges Maschinenprogramm versteht, muß dem Computer mitgeteilt werden, in welchen Adressen das Programm abgespeichert werden soll und welche Adressen als Datenspeicher benutzt werden. Hierfür gibt es gewisse Assembleranweisungen, die je nach Gerät verschieden sind, aber im allgemeinen darauf hinauslaufen, daß nur die Speicherstelle, mit der das Programm beginnen soll, in den Computer einzugeben ist. Für Datenspeicher wird im allgemeinen auch die erste Speicherstelle und dann die Anzahl der für das betreffende Datenfeld zu reservierenden Speicherstellen festgelegt. Das eigentliche Assemblierprogramm arbeitet dann meist in zwei Stufen. Beim ersten Durchlauf des Anwenderprogramms werden alle benutzten symbolischen Adressen ermittelt und in einer Tabelle zusammen mit der sich ergebenden wirklichen Adresse abgespeichert. Im zweiten Durchlauf erfolgt dann die eigentliche Codierung der Befehle mit den im Befehl stehenden Adreßanweisungen, die aus der Tabelle entnommen werden. Es entsteht das *Objektprogramm.*

Die Entwicklungsgeräte erlauben es auch, einzelne Teilprogramme als sogenannte *Programmodule* mit richtiger Adressenfolge zusammenzufassen. Man nennt diesen Vorgang auch *Binden,* wobei der Computer selbsttätig alle im Programm vorkommenden Adressen umrechnet. Auf diese Weise kann man also ein Anwenderprogramm aus schon früher geschriebenen Programmodulen aus einer *Programmbibliothek* zusammenstellen.

Auch das Assemblierprogramm meldet von ihm festgestellte offensichtliche Fehler. Es kann aber natürlich auch nur Fehler, die in falscher Anwendung der Syntax, d.h. der Regeln der Assemblersprache, begründet sind, feststellen, denn kein Computer kann dem Menschen das Denken, also das Bestimmen der richtigen logischen Reihenfolge der einzelnen Befehle, abnehmen. Die Verwendung eines computergesteuerten Programmentwicklungsgerätes kann den Programmierer von der reinen Routinearbeit der Codierung und Adressenverwaltung befreien und dadurch eine ganze Reihe von Fehlermöglichkeiten ausscheiden. Ob ein Programm aber richtig läuft und die erwünschten Ergebnisse bringt, hängt trotzdem einzig und allein vom Können des Programmierers selbst ab.

7.3.6 Prüfen, Testen und Fehler beseitigen

Es wird kaum gelingen, ein etwas längeres Mikroprozessor-Programm so fehlerfrei zu schreiben, daß es auf Anhieb richtig läuft. Man kann durchaus damit rechnen, daß das Austesten eines Programms wenigstens so lange dauert wie die ganze vorherige Programmierarbeit. Ein Anfänger sollte sich nicht wundern, wenn zunächst gar nichts klappt.

Das beste Hilfsmittel für einen Anfänger, ein Programm zu testen, ist der Schrittbetrieb, der bei allen Lern- und Übungsgeräten möglich ist. Bevor man nun allerdings mit dem Testen beginnt, sollte man sich zunächst noch einmal an der Anzeige des Geräts vergewissern, daß alle Befehlscodierungen wirklich richtig eingegeben und vom Gerät richtig in nacheinander folgende Speicherplätze abgelegt worden sind. Ist dies der Fall, so kann man zunächst einmal versuchen, das Programm als Ganzes zu starten und wird dann aber wohl in den meisten Fällen feststellen, daß es noch nicht richtig läuft. Jetzt kommt das eigentliche Prüfen.

Beim Testen muß man systematisch vorgehen. Einer der häufigsten Fehler bei von Hand codierten Programmen sind falsche Sprungzieladressen. Wenn man sein Programm im Schrittbetrieb laufen läßt, kann man recht schnell kontrollieren, ob die vorgesehenen Sprünge richtig ausgeführt werden oder nicht. Meist erkennt man hierbei auch, ob sich irgendwo ein ganz falscher Befehl eingeschlichen hat, dann erscheint ein ganz falscher Wert als nächste Befehlsadresse.

Etwas schwieriger ist der Schrittbetrieb, wenn das Programm Schleifen für die Zeitverzögerung enthält, da diese durch Schrittbetrieb nur mühsam nachzuvollziehen sind. Wenn dies der Fall ist, sollte man die betreffenden Register so laden, daß der Schleifabbruch mit wenigen Befehlen zustandekommt.

Als nächstes muß man dann im Schrittbetrieb die Registerinhalte und insbesondere auch den Inhalt des Programm-Status-Wortes überprüfen, denn auch hier gibt es viele Fehlermöglichkeiten, insbesondere, wenn die Wirkung des Übertrags vorher nicht richtig bedacht wurde.

Hat man durch entsprechendes Ändern und Verbessern ein Programm auf diese Weise im Schrittbetrieb zum Laufen gebracht, so muß es natürlich auch noch im Normalbetrieb getestet werden. Wird im Programm eine Datenverarbeitung vorgenommen, so sollten mehrere sehr unterschiedliche Bitmuster als Daten eingegeben werden, denn das Programm

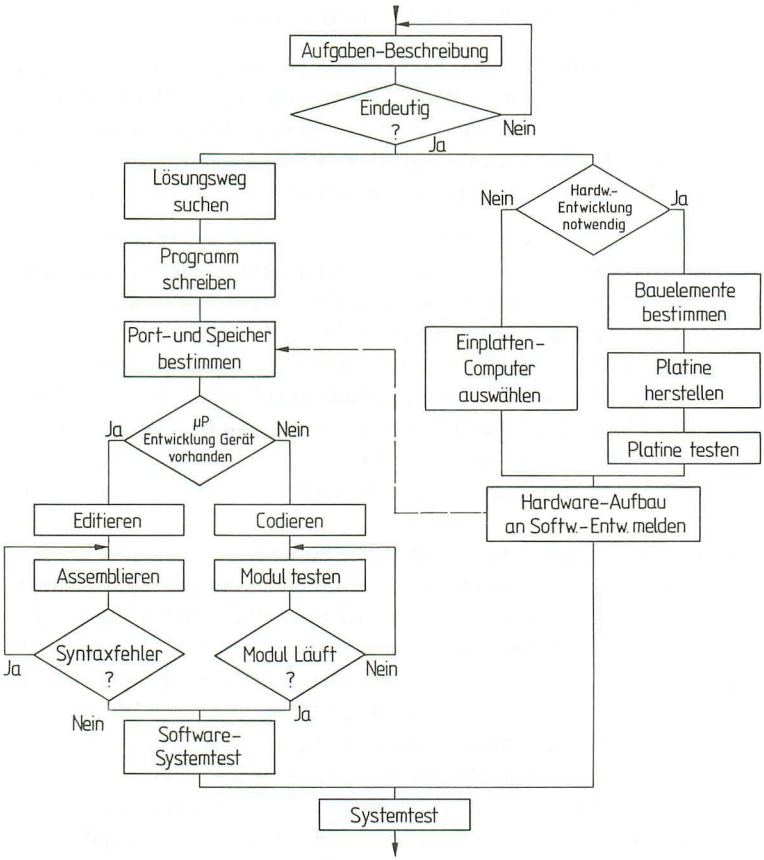


Bild 7.6 Vorgehen bei der Entwicklung eines etwas größeren Mikroprozessor-Programms

kann nur dann als fertig und in Ordnung bezeichnet werden, wenn es alle möglichen Daten richtig verarbeitet.

Eine weitere Möglichkeit, einzelne Programmteile zu testen, stellt der *Haltpunkt* (BREAKPOINT) dar. Wenn man in einem Programm einen Haltpunkt setzt, d.h. eine Adresse bestimmt, an der der Programmlauf anhalten soll, kann man dort den Inhalt aller Register überprüfen und so feststellen, ob das Programm bis dahin zufriedenstellend gelaufen ist. Ist

dies der Fall, muß der Fehler in einem späteren Programmstück liegen, ist dies nicht der Fall, kann man rückwärts immer weitere Haltpunkte setzen, bis der Fehler gefunden ist.

Bei den Lerngeräten wird ein Haltpunkt meist durch Verändern eines Befehls gesetzt. Wenn nun dieser Haltpunkt wegen eines Fehlers im vorhergehenden Programmteil nicht erreicht wird, steht dort im Anwenderprogramm ein falscher Befehl, der erst wieder berichtigt werden muß, wenn das Programm weiterlaufen soll. Bei ordnungsgemäßem Erreichen des Haltpunkts erfolgt die Berichtigung meist selbsttätig durch das Monitorprogramm.

Die Programmentwicklungs-Computer haben meist besondere *Fehlersuchprogramme* (DEBUGGING PROGRAM), die das Fehlersuchen erleichtern. Meist werden durch diese Programme auf einem angeschlossenen Drucker alle Befehlsschritte zusammen mit den sich dabei ergebenden Registerinhalten ausgedruckt, so daß man den Programmablauf im einzelnen verfolgen kann. Hierfür ist natürlich auch eine gute Kenntnis der Maschinensprache notwendig, worauf schon früher hingewiesen wurde.

Nach dem Software-Test des Programms muß nun noch eine weitere Prüfung mit angeschlossener Hardware durchgeführt werden. Auch hierfür bieten die Entwicklungs-Computer gewisse Hilfen, aber hierauf kann in diesem Buch nicht weiter eingegangen werden (Bild 7.6).

7.3.7 Anfertigung der Dokumentation

Jeder Programmierer wird zufrieden sein, wenn das von ihm entwickelte Programm nun endlich richtig läuft. Damit ist seine Arbeit aber noch nicht abgeschlossen, denn unumgänglich ist es, von dem fertigen Programm nun auch eine gute Dokumentation zu erstellen. Während des Programmierens soll man natürlich alle durchgeführten Schritte schriftlich festhalten und nach jeder Änderung das PROGRAM LISTING, also die *Programmliste,* auf den neuesten Stand bringen. Der Ausdruck des kompletten Programms, auch wenn die einzelnen Befehle schon mit entsprechenden Kommentaren versehen sind, genügt aber eigentlich noch nicht als komplette Dokumentation. Hierzu gehört, daß auch die Befehlsbeschreibung und alle Flußdiagramme dem letzten Stand der Entwicklung angeglichen werden. Es muß vor allen Dingen auch genau dokumentiert sein, welche symbolischen Adressen benutzt werden und

wofür die Ein- und Ausgabeleitungen Verwendung finden. Ein klarer Hinweis auf diese Dinge fehlt leider oft auch bei Programmen, die in Fachzeitschriften usw. veröffentlicht werden.

7.4 Strukturierte Programmierung

Die Programmierarbeit kann wesentlich erleichtert werden, wenn ein Programm grundsätzlich aus kleinen, noch überschaubaren Programmelementen, die jeweils nur einen Eingang und einen Ausgang besitzen, aufgebaut wird. Man spricht in diesem Falle auch von einer *strukturierten Programmierung,* bei der die einzelnen Programmelemente, also die Strukturblöcke, hierarchisch in verschiedenen Ebenen, wie es das Bild 7.7 zeigt, angeordnet sind.

Der Vorteil eines derartigen Programmaufbaus, das einem auf dem Kopf stehenden Baum entspricht, liegt vor allen Dingen darin, daß es viel übersichtlicher wirkt als ein Programm mit vielen Übergängen zwischen den einzelnen Programmblöcken durch bedingte oder unbedingte Sprungbefehle, wie es in Bild 7.8 dargestellt ist.

Bei einem gemäß Bild 7.7 in einer Baumstruktur aufgebauten Pro-

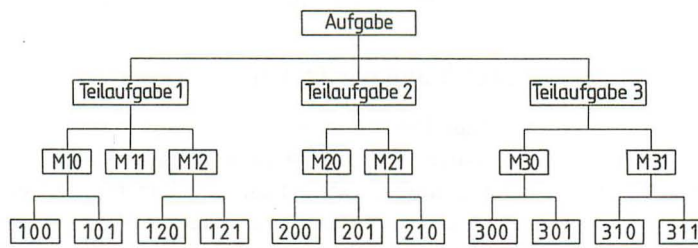


Bild 7.7 Gut strukturiertes Programm mit Verzweigungen bis zur letzten Ebene (Baumstruktur)

Die Hauptaufgabe ist in drei Teilaufgaben zerlegt. Jede dieser Teilaufgaben wiederum wird weiter zerlegt, bis zum Schluß eindeutige, klar überschaubare Programmoduln entstanden sind. Bei jedem Modul besteht nur eine Verbindung zum übergeordneten Modul und eine Verbindung zu jedem untergeordneten Modul. Programmänderungen in den unteren Ebenen wirken sich nicht auf alle anderen Programmteile aus.

Mehrfach benötigte Programmteile müßten dann jedesmal, wenn sie gebraucht werden, als Unterprogramme aufgerufen werden. Für ein Programm, das in dieser Technik erstellt ist, werden weniger Speicherplätze benötigt.

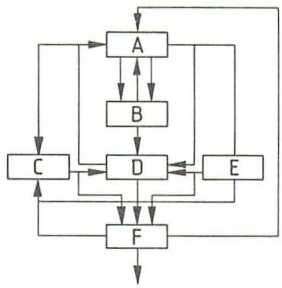


Bild 7.8 Nicht gut strukturiertes Programm
Durch Sprungbefehle zwischen den einzelnen Programmblöcken wird das Programm unübersichtlich. Bei jeder kleinen Änderung kann möglicherweise das ganze Programm beeinflußt werden.

gramm steht auf der obersten hierarchischen Ebene das Hauptprogramm, dessen Aufgabe es eigentlich nur ist, die einzelnen Abschnitte der Unterprogramme der nächsten Ebene anzurufen. In gleicher Weise haben dann wiederum die Programmabschnitte der nächsten Ebene nur Verbindung mit weiteren Unterprogrammen, die für die Erfüllung ihrer eigenen Programmaufgabe notwendig sind.

Wenn aus irgendwelchen Gründen eine Programmänderung in einem der auf unterer Ebene stehenden Unterprogramme notwendig wird, so wird immer nur ein Ast des Baums dadurch beeinflußt, während alle anderen Programmteile ungeändert bleiben können. Wird dagegen eine Änderung in einem Programm, das gemäß Bild 7.8 aufgebaut ist, notwendig, so kann die Änderung in einem Programmteil sehr oft viele andere Programmteile mit beeinflussen.

Zur Erleichterung der strukturierten Programmierung hat man ein besonderes Darstellungssystem für Programmfolgen geschaffen, das man *Struktogramm* nennt. Die einzelnen Elemente dieses Systems sind zum Vergleich mit den Elementen des üblichen Flußdiagramms zusammen auf Bild 7.9 dargestellt. Im Prinzip ist es jedoch gleich, ob man ein Programm in Form eines Struktogramms oder eines Flußschaubilds darstellt.

Zur strukturierten Programmierung gehört es auch, daß man bei der eigentlichen Programmierarbeit nach der *Methode der schrittweisen Verfeinerung* (TOP DOWN PROGRAMMING) arbeitet. Man versteht hierunter ein Programmieren, bei dem man zunächst ganz grob festlegt, was man erreichen will. Dann wird das gewünschte Ziel der Programmierarbeit in einige notwendige Einzelschritte unterteilt. Jeder dieser Einzel-

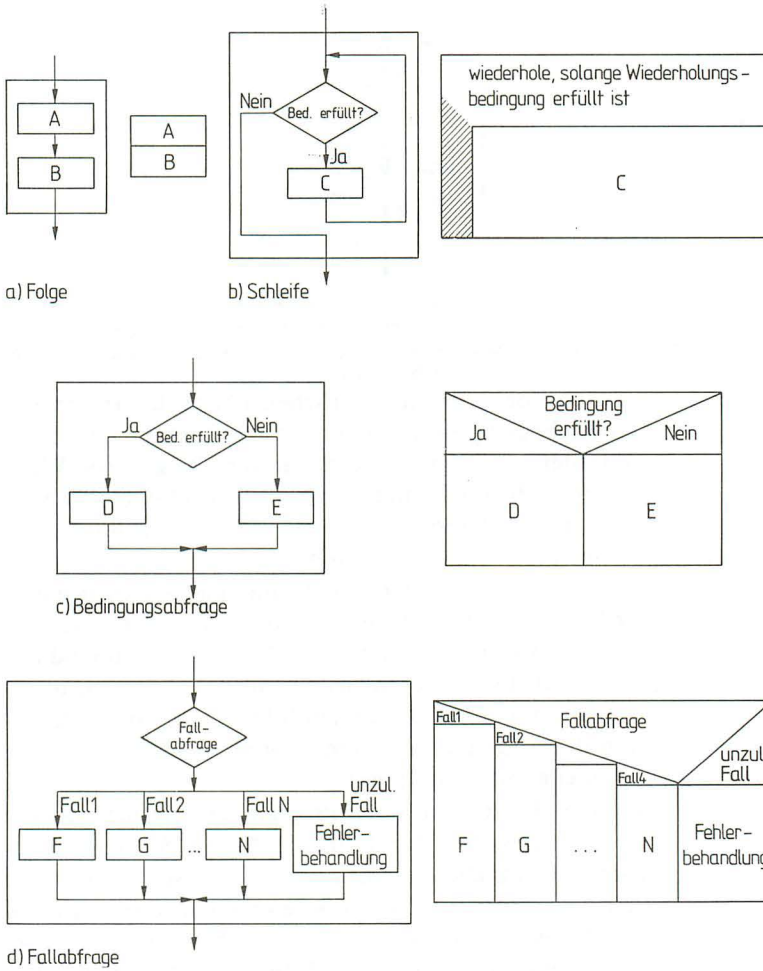


Bild 7.9 Elemente für die Darstellung eines Programmablaufs als Struktogramm

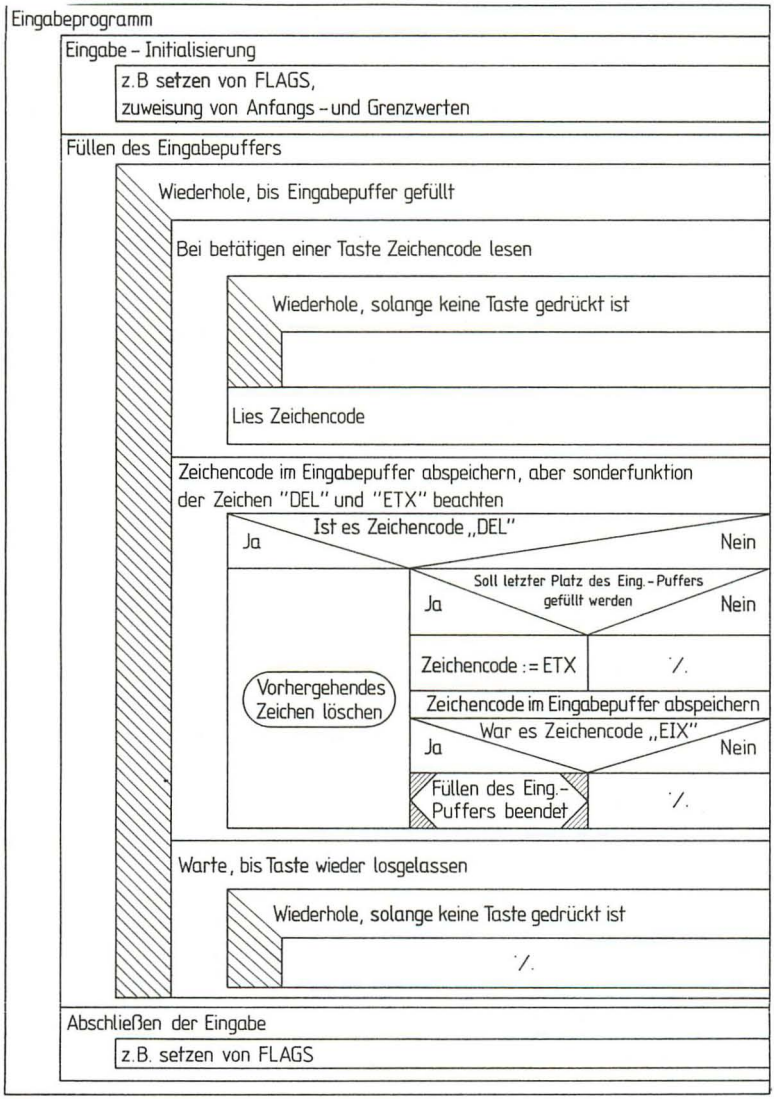


Bild 7.10 Beispiel einer strukturierten Programmierung mit Darstellung als Struktogramm

schritte setzt sich dann wieder aus verschiedenen Programmelementen zusammen, die nach unten hin immer weiter verfeinert werden, bis sie durch den Befehlssatz, der dem Programmierer zur Verfügung steht, ausdrückbar sind. Ein einfaches Beispiel für die Strukturierung eines Programms mit Hilfe der Struktogramm-Darstellung ist in Bild 7.10 gezeigt.

7.5 Programmanalysen

Sehr viel lernen kann man, wenn man versucht, von Experten geschriebene Programme zu analysieren. Dies ist oft nicht ganz einfach, insbesondere, wenn nur ein Assemblerprogramm mit vielen symbolischen Adressen zur Verfügung steht. Voraussetzung für eine Programmanalyse ist es darum, daß entweder das Assembler- und das vollständige Maschinenprogramm bekannt sind oder aber, daß man die Bedeutung und die Zahlenwerte der symbolischen Adressen kennt.

Schwierigkeiten bringt auch die Tatsache, daß bei den meisten Programmbeispielen, die in Zeitschriften usw. veröffentlicht werden, klare Aussagen über die Programmschnittstellen, also die Eingabe- und Ausgabeport, und ihre Bedeutung sowie über die eventuell benötigte Ein- und Ausgabeparameter nicht gemacht werden. Hier gilt es manchmal, etwas Detektivarbeit zu leisten, um zunächst einmal die Voraussetzungen, unter denen der Programmierer gearbeitet hat, herauszufinden.

An dem Beispiel 7.11 soll gezeigt werden, wie man praktisch ein kleines Programm analysieren kann.

Es wird angenommen, daß ein Programm für binäre Multiplikation von zwei 8stelligen Binärzahlen, geschrieben für den *8080*, in Assembler- und Maschinensprache vorliegt, das im einzelnen untersucht und analysiert werden soll.

Das im Bild 7.11a) gezeigte Programm enthält keine symbolischen Adressen, so daß hier keine Schwierigkeiten entstehen. Sollte das Programm symbolische Adressen enthalten, so ist es zweckmäßig, zunächst eine Liste aller Symbole zusammen mit ihrem Zahlenwert anzufertigen.

Wenn es sich um eine etwas kompliziertere Aufgabenstellung handelt, bei der möglicherweise Schleifen mehrfach durchlaufen werden, ist es empfehlenswert, anhand der Befehle des Programms ein Flußschaubild

zu zeichnen. Aus dem vorliegenden Programm ist zu erkennen, daß durch die ersten vier Befehle die Inhalte der Speicherstellen **0430** und **0431** unter Zwischenschaltung des Akkumulators in die Register C und D geladen werden. Man kann ohne weiteres sehen, daß es sich hierbei um die beiden Operanden für den Multiplikationsvorgang handelt.

Durch die nächsten Befehle wird das Register B auf **00** gesetzt und das Register E mit dem Wert **09** geladen. Die Tatsache, daß ein Register auf **00** gesetzt wird, deutet bei arithmetischen Aufgabenstellungen darauf hin, daß dort irgendwelche Ergebnisse oder Zwischenergebnisse abgespeichert werden sollen. Ebenso kann man annehmen, daß das Register E in diesem Fall als Stellenzähler Verwendung findet. Es gibt zwar in einem 8-Bit-Muster nur 8 Stellen, aber bei dieser besonderen Art des Schleifenaufbaus muß man den Zähler um den Wert **1** höher setzen, als es der Zahl der Schleifendurchläufe entspricht.

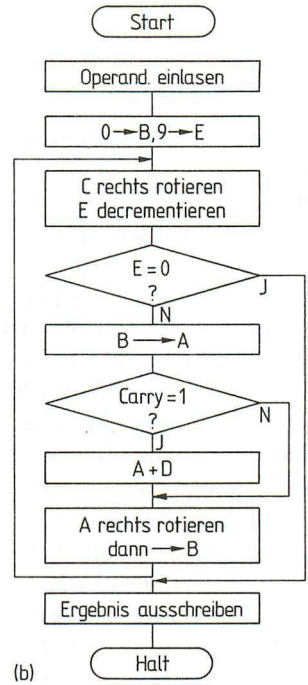
Man erkennt weiterhin zwei bedingte und einen unbedingten Sprungbefehl. Der erste bedingte Sprung erfolgt, wenn das Register E den Wert **00** hat, d.h., wenn acht Schleifen durchlaufen sind, dann muß das Ergebnis ausgegeben werden. Der zweite bedingte Sprungbefehl erfolgt immer dann, wenn das Carry-Bit nicht gesetzt ist. Der unbedingte Sprung führt wieder zum Schleifenanfang.

Die eigentliche Multiplikation erfolgt durch die Rotationsbefehle und den Additionsbefehl. Wenn man diese Befehle analysiert, muß man beachten, daß sich Rotationen und Additionen beim *8080* immer auf den Inhalt des Akkumulators beziehen. Vor dem ersten Rotationsbefehl wurde der Akkumulator mit dem Inhalt des Registers C geladen, dessen Inhalt einschließlich des Carry-Bit dann durch den Befehl um eine Position nach rechts verschoben wird. Danach wird der sich so ergebende neue Inhalt wieder vom Akkumulator in das Register C abgespeichert. Beim zweiten Rotationsbefehl befindet sich der Inhalt des Registers B im Akkumulator, je nachdem, ob bei der ersten Rotation des Inhalts des Registers C das Carry-Bit auf **1** gesetzt wurde oder nicht, erfolgt kein Sprung und nur eine Rotation, oder es wird zunächst der Inhalt des Registers D zum Inhalt des Registers A hinzuaddiert und dann erst die Rotation ausgeführt. Nach dieser Datenverarbeitung wird dann der Inhalt des Akkumulators wieder in das Register B abgespeichert.

Dies alles kann man aus dem Programm ersehen und das entsprechende Flußdiagramm dazu zeichnen, wie es in Bild 7.11b) dargestellt ist. Um aber den Vorgang der Multiplikation verstehen zu können, muß

```
LDA     0430H  0400 3A3004
MOV C, A       0403 4F
LDA     0431   0404 3A3104
MOV D, A       0407 57
MOVI B,00      0408 0600
MOVI E,09      040A 1E09
MOV A, C       040C 79
RAR            040D 1F
MOV C, A       040E 4F
DECR E         040F 1D
JZ      041DH  0410 CA1D04
MOV A, B       0413 78
JNC     0418H  0414 D21804
ADD D          0417 82
RAR            041B 1F
MOV B, A       0419 47
JMP     040CH  041A C30C04
MOV A, B       041D 7B
STA     0432H  041E 323204
MOV A, C       0421 79
STA     0433   0422 323304
HALT           0425 76
```

(a)



(b)

Stellenzähler E	A	1. Operand C	2. Operand D	B	Carry
0 1 0 1		1 1 1 0	1 1 0 0	0 0 0 0	
0 1 0 0		0 1 RR 1 1			0
	0 0 0 0	←		0 0 0 0	0
	0 0 0 0	RR →		0 0 0 0	0
0 0 1 1		0 0 RR 1 1			1
	0 0 0 0	←		0 0 0 0	1
	+1 1 0 0				0
	1 1 0 0	RR →		0 1 1 0	0
0 0 1 0		0 0 RR 0 1		0 1 1 0	1
	0 1 1 0	←		0 1 1 0	1
	+1 1 0 0				1
	0 0 1 0	RR →		1 0 0 1	0
0 0 0 1		0 0 RR 0 0			1
	1 0 0 1	←		1 0 0 1	1
	+1 1 0 0				1
	0 1 0 1	RR →		1 0 1 0	1
0 0 0 0		1 0 RR 0 0			

(d) 1 0 1 0 1 0 0 0

```
12 x 14
  48
 12
 168

1100 x 1110
   11000
  1100
 1100
10101000 =128 + 32 + 8 = 168₁₀
```

(c) Binäre Multiplikation

Bild 7.11 Beispiel für die Anlayse eines Mikroprozessor-Programms
Das vorliegende binäre Multiplikationsprogramm (a) wird zunächst in ein Flußdiagramm (b) umgesetzt, um die verschiedenen Programmschleifen erkennen zu können. Dann wird eine Liste der beteiligten Register, Speicherstellen usw. angefertigt und der Programmablauf Befehl für Befehl nachvollzogen. Dabei kann es vereinfachend sein, zur Verkürzung anstelle mit einem 8-Bit-Muster nur mit einem 4-Bit-Muster zu arbeiten. Bei mathematischen Operationen ist es immer zweckmäßig, anhand bekannter Vorgehensweise im Dezimalsystem auch das Vorgehen im Binärsystem darzustellen (c).

man sich weiteres Anschauungsmaterial schaffen.

Bei der Analyse aller arithmetischen Operationen ist es zweckmäßig, daran zu denken, daß auch das binäre Zahlensystem die gleichen Rechenregeln benutzt, die wir aus dem Dezimalsystem kennen. Man sollte also zunächst einmal überlegen, wie wird dezimal gerechnet, und dies dann auf die Binär-Rechnung übertragen. Dabei ist es grundsätzlich zweckmäßig, die Binär-Rechnung mit möglichst einfachen Zahlen durchzuführen. Da die gleichen Regeln für jedes beliebig lange Bitmuster gelten, kann man das Beispiel ohne weiteres statt mit 8 Bit mit 4 Bit durchrechnen, weil es dann viel übersichtlicher bleibt.

In Bild 7.11c) ist der Rechenvorgang bei schriftlicher Multiplikation dargestellt, wie er auch schon in Kapitel 2 erklärt wurde. Man erkennt, daß die binäre Addition jeweils durch Schiebe- und Additionsvorgänge zurückzuführen ist.

Genau das geschieht auch in dem vorgestellten Programm. Die Schwierigkeit besteht nur darin, daß die Programmverfolgung wegen der durchlaufenen Schleifen recht schwierig ist. Es gibt aber auch hier eine

Methode, wie man sich helfen kann. Man braucht zunächst eine Tabelle mit je einer Spalte für jedes der programmbeteiligten Register, wie dies in Bild 7.11d) dargestellt ist. Auch hier kann man zur Vereinfachung mit 4-Bit-Mustern statt mit 8-Bit-Mustern arbeiten, muß dabei natürlich daran denken, daß in diesem Fall der Schleifenzähler nicht auf **9**, sondern auf **5** zu setzen ist. Jetzt werden die Ausgangswerte der Register in die Tabelle eingetragen und dann nach jedem Befehl die durch den Befehl veränderten Registerinhalte eingesetzt. Wenn dann beim Addieren und Verschieben keine Fehler gemacht werden, was leider bei der Kopfrechnung nicht auszuschließen ist, muß das Programm nach 4 Schleifen zum Ende kommen und das Ergebnis ausgeben. Wie im Beispiel zu sehen ist, gibt es dabei noch die Besonderheit, daß der obere Ergebnisteil im Register B und der untere Ergebnisteil im Register C steht.

Auf ähnliche Weise lassen sich auch andere Programme analysieren, wenn man die Geduld aufbringt, sich wirklich zu überlegen, was bei jedem Programmschritt passiert. Wichtig für alle Programmanalysen ist es, daß man dabei immer den Inhalt der einzelnen Bit des Programm-Status-Wortes mit berücksichtigt und daran denkt, durch welche Befehle es verändert werden kann. Man muß auch darauf achten, in welchen Registern die Datenverarbeitung wirklich stattfindet und wo nur Daten transportiert werden. Bei der Analyse von Programmen, die für den *8080* geschrieben sind, ist immer der Inhalt des Registerpaars HL mit in Betracht zu ziehen. Bei anderen Prozessoren muß man immer den Inhalt des Indexregisters vor Augen haben.

8

Beispiel für das Vorgehen bei einer Programmentwicklung

In diesem Kapitel wird gezeigt, welche Überlegungen bei der Entwicklung eines Programms anzustellen sind, welche Schwierigkeiten und Fehler dabei auftreten können und wie man mit diesen Schwierigkeiten fertig wird.

Da alle Einzelheiten genau beschrieben werden, konnte nur ein verhältnismäßig kurzes Programm gewählt werden, das der Leser auch leicht überblicken und gegebenenfalls nachvollziehen kann. Das Programm ist für den *2650* geschrieben und kann auf dem Lern- und Übungsgerät INSTRUCTOR 50 ausprobiert werden. Wenn ein Leser mit einem anderen Prozessor arbeitet, ist es eine sehr gute Übung, dieses Programm für den anderen Prozessor umzuschreiben.

8.1 Aufgabenstellung

Durch den Mikroprozessor soll ein Würfel, dargestellt durch 7 LED-Anzeigen, betrieben werden. Nach Betätigung einer Taste sollen die geworfenen Augen angezeigt werden. Wahlweise sollen die Augen mehrerer Würfe addiert und auf einer zweistelligen 7-Segment-Anzeige lesbar erscheinen.

Die Aufgabenstellung erscheint im wesentlichen so klar formuliert, daß der Programmierer damit etwas anfangen kann. Bei genauer Durchsicht findet man aber noch zwei Unklarheiten. Es wird von der «Betätigung» einer Taste gesprochen. Es ist also nicht ganz klar formuliert, ob das Würfelergebnis erscheinen soll, wenn die Taste gedrückt oder wenn

sie losgelassen wird. Für dieses kleine Spielprogramm ist die Art der Bedienung nicht so wichtig und kann darum dem Programmierer überlassen werden. Im Gegensatz dazu muß bei allen sonstigen Programmen, insbesondere, wenn dadurch Maschinen oder Apparate gesteuert werden, gerade die Art der Bedienung im voraus durch denjenigen, der die Aufgabe stellt, ganz genau festgelegt werden, denn sonst kann es leicht bei der ersten Vorstellung der Maschine Überraschungen geben, wenn der Programmierer selbst einen Weg gewählt hat, der sich als ungünstig für die tatsächliche Bedienung erweist.

Auch für den zweiten Teil der Aufgabenstellung ist hier diese Bedienung nicht genau festgelegt. Die Bedingung, daß «wahlweise addiert werden soll», kann entweder durch einen Schalter mit zwei definierten Schaltstellungen oder durch Betätigen einer Taste, die beim ersten Druck die Addition einschaltet, beim zweiten Druck diese ausschaltet, erfolgen. In der Aufgabenbeschreibung ist auch nichts darüber gesagt, was erfolgen soll, wenn die Addition ausgeschaltet ist. Soll dann auch die Augenzahl digital angezeigt werden oder soll dann die Anzeige dunkel sein?

Man sieht also schon bei diesem recht einfachen Beispiel, wie wichtig es ist, bereits in der Aufgabenstellung ganz genaue Formulierungen zu verlangen. In diesem Fall soll nun angenommen werden, daß der Programmierer bewußt die am einfachsten zu programmierende Art der Bedienung wählen kann.

8.2 Hardware-Konzept

Verwendet werden soll der Mikroprozessor-Typ *2650*. Dazu gibt es bestimmte Systembausteine. Man kann aber auch beliebige Speicher oder Ein-/Ausgangsbauelemente einschließen.

Die erste Überlegung gilt der Zahl der benötigten Ein- und Ausgabeleitungen. Aus der Aufgabenstellung geht hervor, daß für den Eingang in das System entweder zwei Taster oder ein Taster und ein Schalter gebraucht werden. Als Ausgang werden zunächst 7 Leitungen für die 7 Leuchtdioden des Würfels und außerdem entsprechende Ansteuerleitungen für die zwei 7-Segment-Anzeigen gebraucht.

Man muß sich nun darüber klar werden, wie die 7-Segment-Anzeigen angesteuert werden sollen. Hierfür gibt es verschiedene Möglichkeiten, die hier nicht im einzelnen untersucht werden können, so daß hier

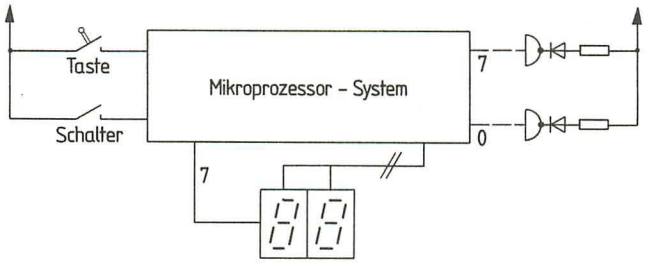


Bild 8.1 Blockschema für das Programm „Würfel mit Anzeige"

angenommen wird, daß zwei Leitungen für die Stellenansteuerung und sieben Segment-Leitungen benötigt werden. Somit ergibt sich für das System ein stark vereinfachtes Blockschaltbild, das im Bild 8.1 dargestellt ist. Hierbei wird berücksichtigt, daß der *2650* einen direkten Eingang in Form des SENSE-Kontaktes besitzt, der für die Taste ausgenutzt wird.

Nun kommt die Abschätzung des Speicherbedarfs. Man kann übersehen, daß das Programm nicht allzu lang werden wird, so daß im Prinzip ein ROM-Speicherbereich mit etwa hundert Speicherstellen ausreichen dürfte. Außerdem werden einige RAM-Speicherstellen gebraucht. Hier reicht der kleinste Baustein mit 8 Bit Breite aus.

8.3 Problemlösung suchen

Als nächstes muß nun ein allgemeiner Lösungsansatz für die Aufgabe gesucht werden. Hierbei bedient man sich am einfachsten der Methode des Flußdiagramms, mit der die sequentiell im Mikroprozessor ablaufenden Vorgänge sehr gut darstellbar sind. Man kann aber auch Struktogramme zeichnen, wie sie im Bild 7.9 dargestellt sind.

Zur Darstellung des eigentlichen Würfelvorgangs kann man sich der Tatsache bedienen, daß ein Mikroprozessor außerordentlich schnell arbeitet, so daß es praktisch unmöglich ist, von außen mit dem Ziel in die Arbeit einzugreifen, ein ganz bewußtes Ergebnis zu erreichen. Man kann den Programmlauf also nur durch Befehle, aber nicht durch Drücken einer Taste stoppen, was die Möglichkeit gibt, durch Programm einen Zufalls-Generator aufzubauen. Wenn man also im Prozessor eine

Zählschleife aufbaut und diese von außen unterbricht, so ist das Ergebnis nicht vorherbestimmbar, sondern nur durch den Zufall bestimmt. Es braucht also nur ein Zähler aufgebaut zu werden, der immer von 1 bis 6 zählt, und wenn dieses Ergebnis erreicht ist, wieder von vorn beginnt.

Zweckmäßig ist es auch, Überlegungen darüber anzustellen, was der Mikroprozessor tun soll, wenn er gerade nichts zu arbeiten hat. Im allgemeinen programmiert man für diesen Zweck eine Warteschleife, in der der Mikroprozessor so lange bleibt, bis ihn ein von außen kommender Befehl veranlaßt, seine eigentliche Arbeit aufzunehmen.

Die einzelnen Arbeitsvorgänge werden zunächst nur als Block dargestellt, und der erste Entwurf für ein beschreibendes Flußdiagramm, in der alle notwendigen Operationen der Reihe nach hintereinander dargestellt sind, könnte wie in Bild 8.2a) gezeigt aussehen.

Wenn man dieses Diagramm aber etwas näher betrachtet, wird man einige Unschönheiten feststellen, denn wie es da gezeichnet ist, kann mit der Taste zwar die Bildung einer Zufallszahl eingeleitet werden, aber es ist nicht möglich, den Zufallszähler beliebig anzuhalten. Man muß also die Aufgabenlösung etwas anders aufbauen, und so entsteht als zweites beschreibendes Flußdiagramm die unter b) gezeigte Lösung.

Wenn bei dieser Lösung die Taste nicht gedrückt ist, so läuft das Programm in einer Schleife mit laufender Erhöhung der Zufallszahl. Wird irgendwann die Taste gedrückt, erfolgt der Abbruch der Schleife, und die gerade gebildete Zufallszahl wird als Würfelergebnis ausgegeben. Danach wird die Entscheidung abgefragt, ob mit Summenbildung oder ohne gearbeitet werden soll. Im ersten Fall erfolgt die Addition und die Anzeige der Summe, im zweiten Fall entweder nur die Anzeige oder entsprechend dem gestrichelten Flußweg ein Löschen der Anzeige, so daß dort gar nichts erscheint.

Wenn auch dieses Flußdiagramm nun schon recht vernünftig aussieht, so sollte man sich doch noch nicht gleich in die Codierarbeit stürzen, sondern zunächst noch einmal genau untersuchen, wie die Schleifen im Programm durchlaufen werden und ob es dabei keine falschen Wege gibt.

Es ist zunächst eine Warteschleife vorgesehen, in der jedesmal die Anzeige angesteuert wird und die Zufallszahl verändert wird, solange die Taste nicht gedrückt ist. Dies ist im Prinzip richtig, denn eine 7-Segment-Anzeige muß mindestens 50mal in der Sekunde angesteuert werden, wenn das Auge sie als nicht flimmernd aufnehmen soll.

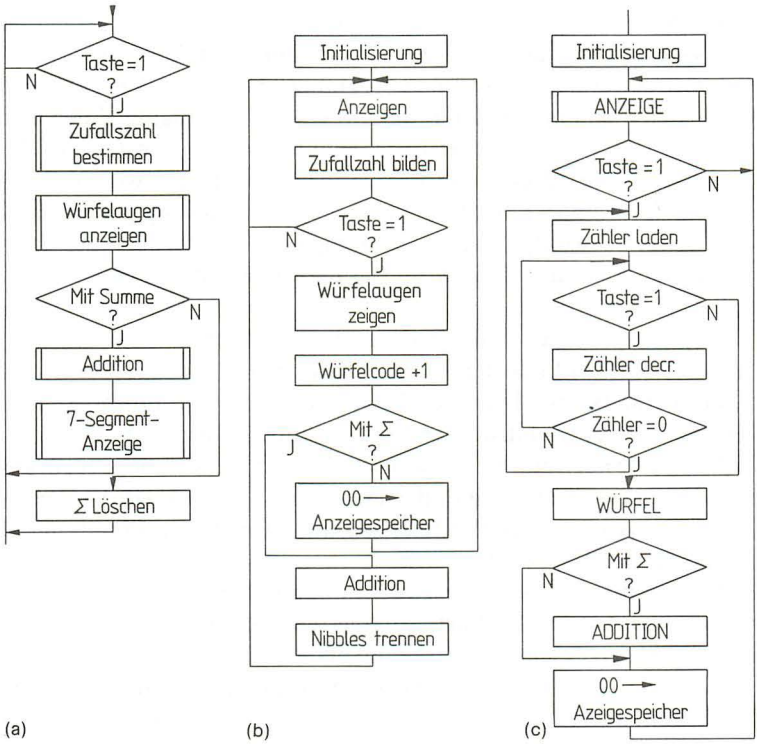


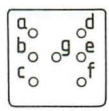
Bild 8.2 Verschiedene Entwürfe für ein beschreibendes Flußdiagramm zum Programmbeispiel „Würfel mit Anzeige"

Wenn man nun allerdings den vorgeschlagenen Lösungsweg weiter untersucht, kommt man zu einer Schwierigkeit. Im Augenblick, wenn die Taste gedrückt wird, springt das Programm in die große Schleife mit der Anzeige der Würfelaugen und gegebenenfalls der Addition. Das Programm würde so einwandfrei laufen, wenn der Mensch in der Lage wäre, die Taste nur so kurz zu drücken, daß die große Schleife nur einmal durchlaufen wird. Da dies aber nicht der Fall ist, würde der Prozessor auch beim noch so kurzen Drücken der Taste die große Schleife mehrfach durchlaufen, es würde jedesmal eine andere Würfelzahl ausgegeben werden, was auch diesen Lösungsweg unbrauchbar macht.

Man muß also versuchen, einen Lösungsweg zu finden, bei dem der Prozessor durch Betätigung eines Schaltorgans veranlaßt wird, die Warteschleife zu verlassen, dann aber die Hauptschleife mit der Ausgabe der Würfelaugen und mit der Addition nur einmal durchläuft. Eine Lösung wäre möglich, wenn man zwei Taster benutzt, was aber unschön ist. Man kann jedoch die Tatsache ausnutzen, daß jede Taste zwei Stellungen besitzt: «Aus» und «Ein». So muß also ein Lösungsweg gefunden werden, bei dem sowohl die «Ein»-Stellung der Taste als auch die «Aus»-Stellung der Taste abgefragt wird. Das entsprechende Flußdiagramm ist in Bild 8.2c) dargestellt.

Die erste Abfrage, in der entschieden wird, ob die Taste gedrückt wurde, erfolgt am Ende der Warteschleife, in der nur die Anzeige angesteuert wird. Ist die Taste gedrückt, geht der Mikroprozessor in die Schleife, die den Zufallszähler betreibt. Innerhalb dieser Zählschleife wird die Taste noch einmal abgefragt. Solange die Taste noch gedrückt ist, läuft der Zähler weiter, wenn die Taste losgelassen ist, erfolgt der Ausprung zur Anzeige der Würfelaugen und zum Additionsteil, der nur einmal durchlaufen wird.

Ehe man nun an die Codierung geht, muß aber noch untersucht werden, wie die Augen des Würfels dargestellt werden sollen. Wenn man einen Würfel betrachtet, so erkennt man, daß er sieben verschiedene Positionen für die Augen hat. Da der Mikroprozessor 8 Bit als Informationsträger zur Verfügung hat, kann also jeder Augenposition eine Bit-



Zufalls Zahl	Bit Nr. → / LED → / Augen ↓	6 a	5 b	4 c	3 d	2 e	1 f	0 g	Hex.	Adresse
0	1	0	0	0	0	0	0	1	01	00A0
1	2	1	0	0	0	0	1	0	42	00A1
2	3	1	0	0	0	0	1	1	43	00A2
3	4	1	0	1	1	0	1	0	5A	00A3
4	5	1	0	1	1	0	1	1	5B	00A4
5	6	1	1	1	1	1	1	0	7E	00A5

Bild 8.3 Würfeltabelle

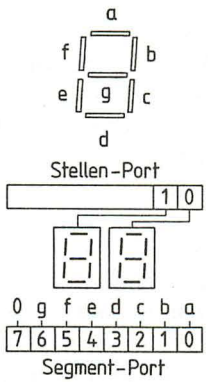
In dieser Tabelle sind die als Würfelergebnis anzuzeigenden Augen als Bit-Muster und in hexadezimaler Codierung dargestellt. Die erste Spalte enthält die jeweils dafür im Programm verwendete Zufallszahl und die letzte Spalte die Adresse, in der das Bit-Muster im Programm abgespeichert ist.

Position zur Verfügung gestellt werden, und es ergibt sich die in 8.3 dargestellte Würfeltabelle.

Das Aufgabenproblem *7-Segment-Anzeige* muß durch eine kleine Voruntersuchung geklärt werden. Die sieben Segmente einer Anzeige sind, wie im Bild 8.4 dargestellt, angeordnet. Man kann nun auch hier leicht eine Tabelle aufstellen, aus der erkennbar wird, welche Elemente leuchten müssen, wenn eine bestimmte Zahl zur Anzeige gebracht werden soll. Auch diese Tabelle, zusammen mit der Hexadezimalcodierung, ist im Bild 8.4 gezeigt.

Die Ansteuerung von 7-Segment-Anzeigen erfolgt vielfach im sogenannten *Multiplex-Verfahren,* was bedeutet, daß die einzelnen Anzeigestellen nicht parallel, sondern in schneller Folge hintereinander angesteuert werden, so daß sie eigentlich immer nur kurzzeitig aufleuchten. Wenn diese Ansteuerung schnell genug hintereinander erfolgt, erscheint die angezeigte Zahl trotzdem flimmerfrei, da das Auge wegen seiner Trägheit das Aufleuchten und das dazwischenliegende Verlöschen gar nicht wahrnehmen kann.

Zur Ansteuerung kann man also alle Segmentleitungen parallel zusammengeschlossen auf ein Port führen und über ein zweites Port jede Stelle der Anzeige ansteuern. Wenn man dann zum Beispiel bei den



Zahl	g	f	e	d	c	b	a	Code
0	0	1	1	1	1	1	1	3F
1	0	0	0	0	1	1	0	0G
2	1	0	1	1	0	1	1	5B
3	1	0	0	1	1	1	1	4F
4	1	1	0	0	1	1	0	6G
5	1	1	0	1	1	0	1	6D
6	1	1	1	1	1	0	1	7D
7	0	0	0	0	1	1	1	07
8	1	1	1	1	1	1	1	7F
9	1	1	0	0	1	1	1	67
A	1	1	1	0	1	1	1	77
B	1	1	1	1	1	0	0	7C
C	0	1	1	1	0	0	1	39
D	1	0	1	1	1	1	0	5E
E	1	1	1	1	0	0	1	79
F	1	1	1	0	0	0	1	71

Bild 8.4 Schema einer 7-Segment-Anzeige mit der dazugehörigen Segmenttabelle

Segmentleitungen für die Segmente, die leuchten sollen, eine **1** ausgibt und am Stellenport nur dort eine **0** ausschreibt, wo die Anzeige erfolgen soll, so leuchten die so angesteuerten Leuchtelemente kurzzeitig auf. Im allgemeinen verwendet man aber für die Ansteuerung von 7-Segment-Anzeigen noch einige zusätzliche Hardware-Bauelemente, und so wird in diesem Beispiel auch angenommen, daß sowohl auf den Segment- als auch auf den Stellenleitungen jeweils eine **1** auszugeben ist, um die Anzeige zum Leuchten zu bringen.

8.4 Programmentwicklung

Für die Programmentwicklung muß man mit überschaubaren *Programmblöcken,* auch *Module* genannt, arbeiten. Bei dem vorliegenden Beispiel kann man leicht drei dieser Module erkennen, und zwar die Programmblöcke:

- *ANZEIGE:* die Warteschleife mit der Anzeige und der ersten Tastaturabfrage.
- *WÜRFEL:* Die Zählschleife für die Zufallszahl mit der zweiten Tastaturabfrage und die Ausgabe der Würfelaugen an die Leuchtdioden.
- *ADDITION:* Abfrage der Additionsentscheidung und Programmteil zur Addition.

Bevor man aber mit der eigentlichen Programmentwicklung für die Module beginnt, sollte man auf einem separaten Blatt eine Liste aller im Programm verwendeten symbolischen Adressen und sonstigen symbolischen Ausdrücke anlegen, wie es im Bild 8.5 dargestellt ist. Man kann diese Liste natürlich noch nicht vollständig zu Beginn der Programmarbeit ausfüllen, da sich einige Adressen erst noch beim eigentlichen Programmieren ergeben, jedoch sollte man alle Adressen, die vorher zu bestimmen sind, dort eintragen.

Da das Beispiel so geschrieben ist, daß es auf einem einfachen Lerngerät, in diesem Fall dem INSTRUCTOR 50, laufen kann, wurde für den Programmbereich die Speicherstellen **0000** bis **00FF** gewählt. Hierbei würde es sich also normalerweise um einen ROM-Bereich handeln, in dem auch die verschiedenen festprogrammierten Tabellen unterzubringen sind. Als RAM-Bereich stehen dann die Speicherstellen von **0100** aufwärts zur Verfügung.

```
Programmstart                    OOOO
RAM                            ab O1OO
Schalter: Addition: PORT D,O1 auf 1 wenn Ja
Taste für Würfel  : SENSE - Eingang
Stellenanwahl        STEPO EQU OOFA
Segmentanwahl     :  SEGPO EQU OOF9
Würfeltabelle        WTAB  EQU OOAO
                           RES    6
Segmenttabelle       SETAB EQU OOA6
                           RES   1O
Anzeigespeicher      ANSP  EQU O1OO
                           RES    2
Summenspeicher       SUM   EQU O1O2
```

WTAB

00A0	01
00A1	42
00A2	43
00A3	5A
00A4	5B
00A5	7E

SETAB

00A6	3F
00A7	06
00A8	5B
00A9	4F
00AA	66
00AB	6D
00AC	7D
00AD	07
00AE	7F
00AF	67

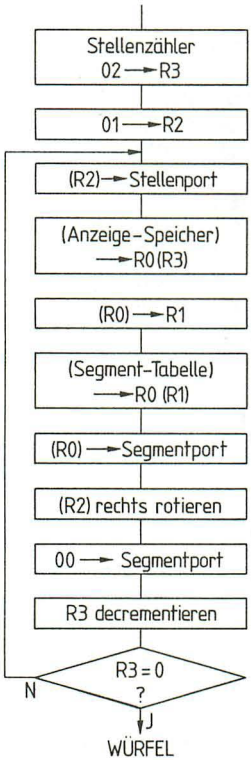
Bild 8.5 Adressenliste für das Programm „Würfel mit Anzeige" mit der Segmenttabelle und der Würfeltabelle

Als Taste für den Würfel wird der Eingang **SENSE** des Prozessors verwendet. Der Wahlschalter wird an die Leitung **01** des Ports **DATA** angeschlossen. Eine dort eingegebene **1** soll bedeuten, daß mit Addition gearbeitet wird.

Für die Stellenanwahl dient beim INSTRUCTOR 50 das erweiterte Port mit der Adresse **FA,** für die Segmentanwahl das Port **F9.**

Vor der Programmierarbeit sollte man dann auch die Würfeltabelle und die Segmenttabelle codieren. Für die Würfeltabelle werden 6 Speicherstellen benötigt und für die Segmenttabelle 10, da ja nur die Ziffern 0 bis 9 anzuzeigen sind.

Da eine 2stellige 7-Segment-Anzeige bedient werden soll, sind hierfür 2 Speicherplätze im RAM-Bereich als Anzeigespeicher notwendig. Außerdem wird ein Summenspeicher gebraucht, für den 1 Speicherplatz ausreicht, wenn man 2 BCD-Zahlen in einem Speicherplatz unterbringt.



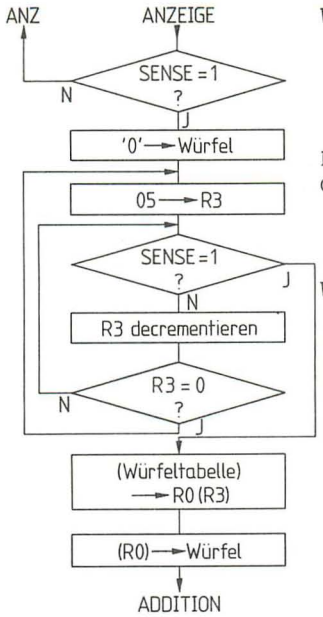

```
ANZEIGE

ANZ   LODI,3 H'02'     0010 0702
      LODI,2 H'01'     0012 0601
MIRA  WRTE,2 STEPO     0014 D6FA
      LODA,0 ANSP,3-   0016 0F4100
      STRZ,1           0019 C1
      LODA,0 SETAB,1   001A 0D60A6
      WRTE,0 SE6PO     001D D4F9
      RRL, 2           001F D2
      EORZ,0           0020 20
      WRTE,0 SE6PO     0021 D4F9
      BRNR,3 MIRA      0023 5B5F
```

Bild 8.6 Entwurf für das Programmodul ANZEIGE

Nun kann mit dem Schreiben des eigentlichen Programms begonnen werden. Für den Anfänger ist es zweckmäßig, wenn er für jedes Programmodul noch ein Befehlsdiagramm in Form eines Flußschaubildes zeichnet, wie sie in den Bildern 8.6, 8.7 und 8.8 dargestellt sind.

Im Prinzip ist es verhältnismäßig gleich, mit welchem der Programmteile man beginnt. Wichtig ist dabei nur, daß man von vornherein berücksichtigt, daß möglicherweise zwischen den einzelnen Programmteilen Übergabeparameter vorgesehen werden müssen. Man versteht hierunter die Übertragung von Informationen von einem Programmteil in einen anderen, was natürlich auch ganz besonders wichtig ist, wenn man mit Unterprogrammen arbeitet. Da in diesem Fall kein Programm-

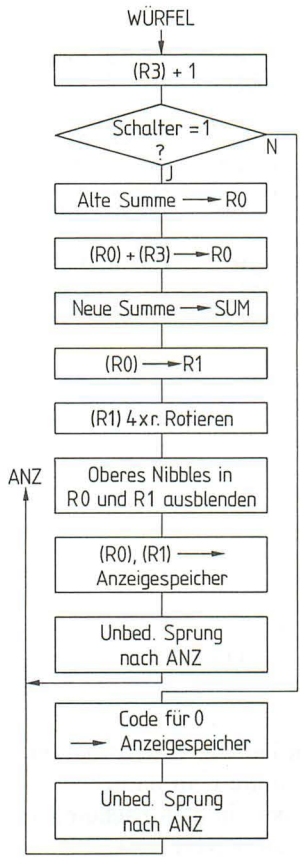



```
WÜRFEL
        TPSU    H'80'     0025 B480
        BCFR,0  ANZ       0027 9867
        EORZ,0            0029 20
        WRTD,0            002A FO
NEU     LODI,3  H'05'     002B 0705
OLGA    TPSU    H'80'     002D B480
        BCTR,0  WURF      002F 9804
        BDRR,3  OLGA      0031 FB7A
        BCTR,3  NEU       0033 1B76
WURF    LODA,0  WTAB,3    0035 OF60A0
        WRTD,0            0038 FO
```

Bild 8.7 Entwurf für das Programmodul WÜRFEL

teil mehrfach verwendet wird, soll in diesem Fall die Unterprogrammtechnik nicht benutzt werden. Im Prinzip könnte man natürlich auch hier ein ganz kurzes Hauptprogramm schreiben, in dem nacheinander nur die drei Programmteile als Unterprogramm angerufen werden.

Es soll nun die Programmierung mit dem Teil **ANZEIGE** begonnen werden. Das Flußdiagramm ist sehr einfach, und man beginnt nun, zunächst die Befehle im Assemblerausdruck niederzuschreiben. Hierbei muß man überlegen, welche Register verwendet werden sollen, so daß es vielleicht zweckmäßig ist, sich hierüber Notizen zu machen. In diesem Fall dient das Register 3 dabei als Indexregister für die Abfrage der Stellen des Anzeigespeichers, in dem die anzuzeigenden Werte als Hexadezimalzahl stehen sollen. Das Register 2 dient zur Ausgabe an das Stellenport, und das Register 1 wird wieder als Index verwendet, um aus der Segmenttabelle die richtige Codierung für die Segmente zu holen. Das Register 0 dient bei den beiden indexierten Befehlen als Zielregister




```
ADDITION
          ADDI,3 H'01'     0038 8701
          REDD,1           003B 71
          TMI, 1 H'01'     003C F501
          BCFR,0 OHNE      003E 981C
          LODA,0 SUM       0040 0C0102
          ADDI,0 H'66'     0043 8466
          ADDZ,3           0045 83
          DAR, 0           0046 94
          STRA,0 SUM       0047 CC0102
          STRZ,1           004A C1
          RRR, 1           004B 51
          RRR, 1           004C 51
          RRR, 1           004D 51
          RRR, 1           004E 51
          ANDI,0 H'0F'     004F 440F
          ANDI,1 H'0F'     0051 450F
          STRA,0 ANSP      0053 CC0100
          STRA,1 ANSP+1    0056 CC0101
          BCTA,3 ANZ       0059 1F0010
OHNE      LODI,0 H'3F'     005C 043F
          STRA,0 ANSP      005E CC0100
          STRA,0 ANSP+1    0061 CC0101
          BCTA,3 ANZ       0064 1F0010
```

Bild 8.8 Entwurf für das Programmodul ADDITION

und dann als Quellregister für die Ausgabe des Segmentcodes an das Segmentport. Am Ende dieses Programmoduls wird an das Segmentport noch einmal der Wert **00** zum Löschen der alten Anzeige ausgegeben, so daß dort immer wieder eine neue Information eingeschrieben werden kann.

In diesem Programmteil werden alle 4 Register gebraucht, eine Informationsübergabe vom 1. Programmteil zum nächsten Programmteil ist nicht notwendig.

Der Programmteil **WÜRFEL** beginnt mit der Abfrage der Taste. Ist diese nicht gedrückt, wird nach **ANZ** zurückgesprungen. Ist die Taste gedrückt, so wird zunächst die alte Leuchtdiodenanzeige durch Ausgabe des Wertes **00** an das **DATA**-Port gelöscht und das Register 3 als Zufallszähler mit **05** geladen. Da die Augen des Würfels von **1** bis **6** gehen, könnte man annehmen, der Zähler müßte auch von **1** bis **6** laufen. Einfacher wird das Programm aber, wenn der Zähler von **0** bis **5** zählt, da man dann den Null-Stand besser abfragen kann. Wie aus der Tabelle im Bild 8.5 zu sehen, bereitet es keine Schwierigkeit, einer bestimmten Zählerzahl eine bestimmte Würfelzahl zuzuordnen.

Es folgt dann die zweite Tastaturabfrage. Der Sprung zur Ausgabe der Würfelaugen erfolgt nur, wenn die Taste losgelassen wurde.

Bei der symbolischen Adresse **WURF** wird dann der Würfelcode mit indexierter Adressierung unter Benutzung des Registers 3 als Index in das Register 0 geladen und dann dieser Wert an das **DATA**-Port ausgegeben, wodurch die Leuchtdioden, die Würfelaugen darstellen, angesteuert werden.

Jetzt ist noch der dritte Programmteil **ADDITION** zu erstellen. Nach dem zweiten Programmteil steht die als Würfelaugenzahl ausgegebene Zufallszahl noch im Register 3. Die dort stehende Zahl ist jedoch um den Wert **1** kleiner als die wirkliche Zahl der Würfelaugen, und darum muß der Inhalt des Registers 3 noch um den Wert **1** für die weitere Verarbeitung erhöht werden.

Jetzt folgt die Abfrage des Schalters, durch den bestimmt wird, ob addiert werden soll oder nicht. Wenn der Schalter auf **0** steht, wird zum Programmteil, der an der symbolischen Adresse **OHNE** beginnt, gesprungen. Dort wird der Wert **00** an die beiden Stellen des Anzeigespeichers ausgegeben, und es erfolgt ein Rücksprung mit absoluter Adressierung zur symbolischen Adresse **ANZ.**

Steht der Wahlschalter auf **1,** wenn also addiert werden soll, muß zunächst eine vorher schon gebildete Summe aus der Speicherstelle **SUM** in das Register 0 geladen werden, damit dort die Addition erfolgen kann. Da nur Dezimalzahlen verarbeitet und angezeigt werden sollen, ist es zunächst notwendig, die Addition auch dezimal zu programmieren, wobei man beim *2650* zunächst den Wert **66** zum ersten Summanden hinzuaddieren muß. Dann folgt die Addition des zweiten Summanden, der immer noch im Register 3 steht, und der Befehl zur Dezimalkorrektur, worauf die Summe abgespeichert werden kann.

Nun gilt es, eine zweite Schwierigkeit zu überwinden, die darin besteht, daß die Summe in gepackter Darstellung, d.h. mit zwei Zahlenwerten in einem Byte zur Verfügung steht, während für die Anzeige eine Darstellung mit nur einem Zahlenwert in einem Byte benötigt wird, damit die Segmenttabelle richtig ausgelesen werden kann. Man muß also die beiden Nibbles der Summe auf 2 Byte aufteilen.

Hierzu wird das noch im Register 0 stehende Ergebnis auch in das Register 1 überschrieben und dieses viermal nach rechts rotiert. Wenn man jetzt bei beiden Registern das obere Nibble ausblendet, liegt der Wert, der anzuzeigen ist, in der gewünschten Form vor.

Damit ist die erste Niederschrift des Programms beendet, ein Vorgang, den man auch *Editieren* nennt, und es geht weiter mit dem *Codieren* des Programms, was, wenn es mit Hilfe eines Computers geschieht, als *Assemblieren* bezeichnet wird.

Zum Assemblieren gehört die Codierung der einzelnen Befehle in Maschinensprache und die richtige Adreßzuweisung. Für größere Programme sollte man hierfür immer einen Computer zur Hilfe haben. Hier wird gezeigt, wie man ein kleines Programm auch ohne Computerhilfe entwickeln kann.

Zunächst wird für die einzelnen Befehle aus der Befehlstabelle des Mikroprozessor-Herstellers die Hexadezimalzahl für den jeweiligen Operationscode herausgesucht. Soweit es sich um Befehle mit absoluter Adressierung handelt, ist die Operandenadresse der Adressenliste zu entnehmen und in richtiger Weise in den Befehl einzusetzen. Soweit in den Befehlen relative Verschiebungen vorkommen, müssen diese berechnet oder aus einer Tabelle abgelesen werden. Wenn man bei Sprungbefehlen die Sprungadresse noch nicht kennt, ist es zweckmäßig, das zweite Befehls-Byte im Sprungbefehl zunächst durch einen Unterstreichungsstrich zu kennzeichnen, damit es bei der Zählung der Befehlsadressen nicht vergessen wird.

Nach der Befehlscodierung werden nun die Befehlsadressen festgelegt. Dabei sollte man auf dem ersten Programmstück nicht von vornherein die Adresse **0000** zuweisen, denn es kann sich beim Prüfen immer wieder ergeben, daß noch einige Befehle vorausgeschickt werden müssen. Im übrigen hat der Programmierer Freiheit, in welchem Teil des Adreßraums er sein Programm anordnen will.

Es bleibt dem Programmierer auch überlassen, ob er, wenn er einmal beim Codieren ist, alle Programmoduln codiert und dann erst mit dem

Prüfen beginnt, oder ob er sich nur ein Modul vornimmt, dieses codiert und dann gleich prüft, denn in jedem Falle ist es sinnvoll, jedes der Programmoduln einzeln durchzuprüfen.

8.5 Prüfen der Programmoduln

Um die einzelnen Abschnitte eines Programms prüfen zu können, muß man klar definierte Anfangs- und Endbedingungen schaffen. Vor dem Prüfen des eigentlichen Programms müssen also die dabei benötigten Speicherstellen mit einfachen und nachprüfbaren Werten geladen werden. Ebenso muß man dafür sorgen, daß man am Ende des Programms erkennt, ob die gewünschte Aufgabe erfüllt wurde. Gegebenenfalls müssen also dem zu prüfenden Programm noch einige Lade- oder Speicherbefehle vor- oder nachgeschaltet werden. Wenn man mit einem Mikroprozessor-Lerngerät arbeitet, ist es meistens möglich, außerhalb des Anwenderprogramms mit Hilfe des Monitors Speicherstellen und Register zu laden, was man gut für die Prüfung des Anwenderprogramms ausnutzen kann.

Das Programm wird dann auch wieder mit Hilfe des Monitors in den RAM-Speicher des Lerngeräts geladen. Anschließend überzeugt man sich zweckmäßigerweise, daß auch alle Befehle richtig eingetastet wurden, und nun kann man, wenn man will, zum erstenmal versuchen, ob das Programm richtig läuft.

Meistens wird man feststellen, daß dies nicht der Fall ist, und dann gilt es, den Fehler zu suchen. Einige Methoden hierfür wurden bereits im Abschnitt 7.3.5 beschrieben, aber manchmal muß man viel Spürsinn aufwenden, um festzustellen, warum ein Programm nicht laufen will. Auf keinen Fall sollte man dabei dem Mikroprozessor die Schuld geben, denn dieser arbeitet immer genauso, wie er programmiert wurde, oder er funktioniert überhaupt nicht.

Wenn bei der Prüfung der einzelnen Programmoduln bewiesen wurde, daß sie ihre Aufgabe erfüllen, erfolgt die Zusammenstellung zum eigentlichen Programm. Es kann möglich sein, daß hierfür noch einige Übergabebefehle zu schreiben sind. Vor allen Dingen muß man aber, ehe man versucht, das Programm als Ganzes zum Laufen zu bringen, noch einmal überprüfen, ob nicht beim Übergang von einem Modul zum nächsten in den benutzten Registern noch unerwünschte Inhalte stehen, die dann

den nächstfolgenden Programmteil verfälschen können. Man sollte also das Programm noch einmal ganz sorgfältig Befehl für Befehl durchgehen und kritisch untersuchen.

Dann wird das Programm hintereinander in der richtigen Reihenfolge geschrieben, wobei möglicherweise die Befehlsadressen neu festzulegen sind. Hierdurch können sich dann auch die Adressen für die Sprungziele ändern, so daß Korrekturen in den betreffenden Sprungbefehlen notwendig sind. Man wird also, wenn das Programm nicht auf Anhieb läuft, auch das Gesamtprogramm entweder im Ganzen oder an kritischen Stellen im Schrittbetrieb überprüfen. Niemand sollte sich wundern, wenn auch jetzt noch wieder Fehler entdeckt werden.

8.6 Endprüfung und Dokumentation

Das Programm wird grundsätzlich zunächst auf einem Lerngerät oder einem Entwicklungs-Computer geprüft, wobei die später anzuschließenden Peripheriegeräte in geeigneter Form simuliert werden. Wenn dort das Programm richtig läuft, kommt der abschließende Test mit der echten Hardware, wobei meistens noch einmal wieder Fehler ausgemerzt werden müssen.

Der Abschluß der Programmentwicklung ist dann die Dokumentation. Zunächst einmal sollte man es sich zur Übung machen, auch alle Entwürfe, Nebenrechnungen, Tabellen usw., soweit sie sich nicht als falsch erwiesen haben, in einem speziellen Programmordner mit aufzuheben. Auf alle Fälle muß daraus der eigentliche Lösungsweg, der gewählt wurde, und die verwendeten Adressen für die Ein- und Ausgänge, Zwischenspeicherplätze usw. erkennbar sein.

Die Programmniederschrift sollte dann einen klaren Einleitungskommentar mit Kurzdarstellung der Programmaufgabe und weitere Kommentare, jeweils wenn sie nötig sind, enthalten. Wenn mit einem Mikroprozessor-Entwicklungsgerät gearbeitet wird, können diese Kommentare dort mitprogrammiert werden, so daß sie bei jedem Ausdruck der Programmliste mit erscheinen. Außer den Überschriftskommentaren, die auch zwischen den einzelnen Programmteilen eingefügt werden können, kann man auf jeder Programmzeile noch einen kurzen Kommentartext geben. In diesem Text sollte nicht stehen, was der Befehl macht, denn dies kann jeder Programmierer aus dem Befehlscode ersehen, son-

```
Programm: Würfel mit Anzeige

Anzeige der Augen eines Würfels, wenn die gedrückte SENSE-Taste
losgelassen wird. Wahlweise Addition der Augenzahl mit Ausgabe
des Ergebnisses aus einer 7-Segment-Anzeige.

      NOP              0000 CO
      NOP              0001 CO          Zum Auffüllen
      PPSL   H'08'     0002 7708        Mit Carry
      CPSL   H'01'     0004 7501        Carry löschen
      EORZ,0           0006 20
      STRA,0 SUM       0007 CC0102 }
      STRA,0 ANSP      000A CC0100 } Speicher löschen
      STRA,0 ANSP+1    000D CC0101 }
ANZ   LODI,3 H'02'     0010 0702        Stellenzähler laden
      LODI,2 H'01'     0012 0601   }
MIRA  WRTE,2 STEPO     0014 D6FA   } Stelle anwählen
      LODA,0 ANSP,3-   0016 0F4100      Anzeige Zahl holen
      STRZ,1           0019 C1          Inh.-Res 0 retten
      LODA,0 SETAB,1   001A 0D60A6      Segmentmuster holen
      WRTE,0 SEGPO     001D D4F9        Seg.-Muster ausgeben
      RRL, 2           001F D2          Nächste Stelle
      EORZ,0           0020 20     }
      WRTE,0 SEGPO     0021 D4F9   } Anzeige löschen
      BRNR,3 MIRA      0023 5B6F        Spr., wenn n. nicht 2. Stelle
      TPSU   H'80'     0025 B480        SENSE prüfen
      BCFR,0 ANZ       0027 9867        Rückspr., wenn nicht gedrückt
      EORZ,0           0029 20     }
      WRTD,0           002A F0     } Würfel Augen löschen
NEU   LODI,3 H'05'     002B 0705        Zufallszähler laden
OLGA  TPSU   H'80'     002D B480        SENSE prüfen
      BCTR,0 WURF      002F 9804        Wenn nicht gedrückt n. WURF
      BDRR,3 OLGA      0031 FB7A        Nächste Zufallszahl
      BCTR,3 NEU       0033 1B76        Neue Zählung
WURF  LODA,0 WTAB,3    0035 0F60A0      Würfelcode holen
      WRTD,0           0038 F0          Augen ausgeben
      ADDI,3 H'01'     0039 8701        Augenzahl korrigieren
      REDD,1           003B 71          Schalter abfragen
      TMI, 1 H'01'     003C F501        Ist Schalter 1?
      BCFR,0 OHNE      003E 981C        Sprung, wenn ohne Summe
      LODA,0 SUM       0040 0C0102      Alte Summe holen
      ADDI,0 H'66'     0043 8466   }
      ADDZ,3           0045 83     } Dezimale Addition
      DAR, 0           0046 94     }
      STRA,0 SUM       0047 CC0102      Wieder speichern
      STRZ,1           004A C1     }
      RRR, 1           004B 51     }
      RRR, 1           004C 51     } Nibbles Trennung
      RRR, 1           004D 51     }
      RRR, 1           004E 51     }
      ANDI,0 H'0F'     004F 440F   }
      ANDI,1 H'0F'     0051 450F   } Obere Teile ausblenden
      STRA,0 ANSP      0053 CC0100 } Summenwert nach
      STRA,1 ANSP+1    0056 CC0101 } Anzeigespeicher
      BCTA,3 ANZ       0059 1F0010      Unbed. Rücksprung
OHNE  LODI,0 H'3F'     005C 043F        Segm.-Code für 0
      STRA,0 ANSP      005E CC0100      in der Anzeige
      STRA,0 ANSP+1    0061 CC0101      Speicher laden
      BCTR,3 ANZ       0064 1F0010      Unbed. Rücksprung
```

Bild 8.9 Programmliste für das Programm „Würfel mit Anzeige" mit dazugehörigen Kommentaren

dern es muß dort stehen, was in bezug auf die gestellte Aufgabe durch den Befehl bewirkt wird.

Die Programmliste für das besprochene Programm ist im Bild 8.9 zu sehen. Man erkennt, daß den drei besprochenen Programmoduln noch ein kurzer Programmteil zur Initialisierung vorgeschaltet wurde. Damit das Programm an der Adresse **0000** gestartet werden kann, erhält es ganz am Anfang vier **NOP**-Befehle. Dadurch wurde vermieden, daß die vorher schon festgelegte Adreßnumerierung noch geändert werden muß.

Dieses kleine Beispiel, dessen Aufgabe zunächst sehr einfach erschien, zeigt bereits, wieviele Überlegungen notwendig sind, um ein Mikroprozessor-Programm zu erstellen. In diesem Buch wurden die Grundbegriffe hierfür erklärt, und es wurde gezeigt, wie man sich mit der Programmiertechnik vertraut machen kann.

Nicht jeder, der mit Mikroprozessoren zu tun hat, muß nun selbst programmieren. Für viele wird es genügen, zu wissen, wie programmiert wird, um dadurch in der Lage zu sein, Aufgabenstellungen, die mit Mikroprozessoren gelöst werden sollen, richtig zu formulieren. Wer jedoch selbst programmieren will, muß sich die hierfür notwendige Erfahrung durch Üben aneignen. Hierfür wünscht der Verfasser guten Erfolg.

9
Anhang

ASCII-Code in hexadezimaler Darstellung (höchstwertigstes Bit = 0)

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	\	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	26	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5\	Z	6A	J	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	58	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	–	3D	=	4D	M	5D	]	6D	m	7D	}
0E	S0	1E	RS	2E	·	3E	>	4E	N	5E	∧↑	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	–←	6F	a	7F	DEL

Vergleichstabelle: Hexadezimal — Dezimal

6. Stelle		5. Stelle		4. Stelle		3. Stelle		2. Stelle		1. Stelle	
HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	04,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	54,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

Hex-Dez-Wandlung:
Addition der Dez-Werte, die den einzelnen Hex-Stellen entsprechen

Beispiel $423A_{16}$ =

$$\begin{array}{r} 16\,384 \\ +\quad 512 \\ +\quad 48 \\ +\quad 10 \\ \hline 16\,954_{10} \end{array}$$

Dez-Hex-Wandlung:
Dez-Wert vom höchstmöglichen Hex-Wert subtrahieren ergibt höchste Hex-Stelle. Subtraktion des Restes gibt nächste Hex-Stelle usw.

Beispiel $50\,100_{10}$

$$\begin{array}{r} 50\,100 \\ -49\,152 \\ \hline 948 \\ -\quad 768 \\ \hline 180 \\ -\quad 176 \\ \hline 4 \end{array}$$

$C\,3\,B\,4_{16}$

Tabelle zur Ermittlung relativer Verschiebungen

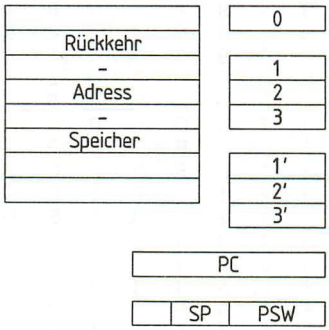
		Nächste Adresse															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Sprung-Adresse	0	0	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1
	1	1	0	F	E	D	C	B	A	9	8	7	6	5	4	3	2
	2	2	1	0	F	E	D	C	B	A	9	8	7	6	5	4	3
	3	3	2	1	0	F	E	D	C	B	A	9	8	7	6	5	4
	4	4	3	2	1	0	F	E	D	C	B	A	9	8	7	6	5
	5	5	4	3	2	1	0	F	E	D	C	B	A	9	8	7	6
	6	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8	7
	7	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8
	8	8	7	6	5	4	3	2	1	0	F	E	D	C	B	A	9
	9	9	8	7	6	5	4	3	2	1	0	F	E	D	C	B	A
	A	A	9	8	7	6	5	4	3	2	1	0	F	E	D	C	B
	B	B	A	9	8	7	6	5	4	3	2	1	0	F	E	D	C
	C	C	B	A	9	8	7	6	5	4	3	2	1	0	F	E	D
	D	D	C	B	A	9	8	7	6	5	4	3	2	1	0	F	E
	E	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	F
	F	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

Die nächsthöhere Stelle ist um −1 zu korrigieren, wenn die Ablesung oberhalb der Diagonale erfolgt ist.

Darstellung von positiven und negativen Dezimalzahlen, im Zweier-Komplement hexadezimal codiert

DEZ	HEX	2 KPL	DEZ	HEX	2 KPL	DEZ	HEX	2 KPL	DEZ	HEX	2 KPL	DEZ	HEX	2 KPL
00	00	00	26	1A	E6	52	34	CC	78	4E	B2	104	68	98
01	01	FF	27	1B	E5	53	35	CB	79	4F	B1	105	69	97
02	02	FE	28	1C	E4	54	36	CA	80	50	B0	106	6A	96
03	03	FD	29	1D	E3	55	37	C9	81	51	AF	107	6B	95
04	04	FC	30	1E	E2	56	38	C8	82	52	AE	108	6C	94
05	05	FB	31	1F	E1	57	39	C7	83	53	AD	109	6D	93
06	06	FA	32	20	E0	58	3A	C6	84	54	AC	110	6E	92
07	07	F9	33	21	DF	59	3B	C5	85	55	AB	111	6F	91
08	08	F8	34	22	DE	60	3C	C4	86	56	AA	112	70	90
09	09	F7	35	23	DD	61	3D	C3	87	57	A9	113	71	8F
10	0A	F6	36	24	DC	62	3E	C2	88	58	A8	114	72	8E
11	0B	F5	37	25	DB	63	3F	C1	89	59	A7	115	73	8D
12	0C	F4	38	26	DA	64	40	C0	90	5A	A6	116	74	8C
13	0D	F3	39	27	D9	65	41	BF	91	5B	A5	117	75	8B
14	0E	F2	40	28	D8	66	42	BE	92	5C	A4	118	76	8A
15	0F	F1	41	29	D7	67	43	BD	93	5D	A3	119	77	89
16	10	F0	42	2A	D6	68	44	BC	94	5E	A2	120	78	88
17	11	EF	43	2B	D5	69	45	BB	95	5F	A1	121	79	87
18	12	EE	44	2C	D4	70	46	BA	96	60	A0	122	7A	86
19	13	ED	45	2D	D3	71	47	B9	97	61	9F	123	7B	85
20	14	EC	46	2E	D2	72	48	B8	98	62	9E	124	7C	84
21	15	EB	47	2F	D1	73	49	B7	99	63	9D	125	7D	83
22	16	EA	48	30	D0	74	4A	B6	100	64	9C	126	7E	82
23	17	E9	49	31	CF	75	4B	B5	101	65	9B	127	7F	81
24	18	E8	50	32	CE	76	4C	B4	102	66	9A	128		80
25	19	E7	51	33	CD	77	4D	B3	103	67	99			

2650



CPU-Register

PSU 7 6 5 X X 2 1 0 — Oberes Programm - Status-Wort

- SP = Stapel Zeiger
- II = Unterbrechungs Sperre = "1"
- F = FLAG Direkter Ausgang
- S = SENSE Direkter Eingang

PSL 7 6 5 4 3 2 1 0 — Unteres Programm - Status-Wort

- C = CARRY Übertrag
- COM = 1 = Log. 0 - Arithm. Vergleich
- OVF = Bereichsüberschreitung
- WC = Mit Carry = "1"; Ohne Carry = "0"
- RS = "0" = Reg. Bk. 0; "1" = Reg. Bk. 1
- IDC = Übertrag von Bit 3 auf Bit 4
- CC = Bedingungscode

Indexkontrolle	
0 0	Kein Index
0 1	IX + 1
1 0	IX − 1
1 1	Nur Index
Adr. + (Rx) = Adr.	

C I	C O	setzen	Vergleichen
0	0	NULL	GLEICH
0	1	POS.	GRÖSSER IX
1	0	NEG.	KLEINER
0	0	Alle Bit = 1	
1	0	Nicht alle Bit = 1	

2650 Befehlstabelle 1

	Assembler	Register 0	1	2	3	C	IDC	OVF	CC	Byte	Befehlsablauf
Laden – Speichern	LODZ	–	01	02	03				●	1	(R) → RO
	LODI	04	05	06	07				●	2	Wert → R
	LODR	08	09	0A	0B				●	2	(PC+V) → R
	LODA	0C	0D	0E	0F				●	3	(Adr.) → R
	STRZ	–	C1	C2	C3				●	1	(RO) → R
	STRR	C8	C9	CA	CB					2	(R) → (PC+V)
	STRA	CC	CD	CE	CF					3	(R) → Adr.
Ein-/Ausgabe	EDDD	70	71	72	73				●	1	(Port D) → R
	EDDC	30	31	32	33				●	1	(Port C) → R
	EDDE	54	55	56	57				●	2	(Ext. Port) → R
	WRTD	F0	F1	F2	F3					1	(R) → Port D
	WRTC	B0	B1	B2	B3					1	(R) → Port C
	WRTE	D4	D5	D6	D7					2	(R) → Ext. Port
Rechnen	ADDZ	80	81	82	83	●	●	●	●	1	(RO) + (R) → RO
	ADDI	84	85	86	87	●	●	●	●	2	(R) + Wert → R
	ADDR	88	89	8A	8B	●	●	●	●	2	(R)+(PC+V) → R
	ADDA	8C	8D	8E	8F	●	●	●	●	3	(R) + (Adr.) → R
	SUBZ	A0	A1	A2	A3	●	●	●	●	1	(RO)−(R)→RO
	SUBI	A4	A5	A6	A7	●	●	●	●	2	(R)−Wert→R
	SUBR	A8	A9	AA	AB	●	●	●	●	2	(R)−(PC+V)→R
	SUBA	AC	AD	AE	AF	●	●	●	●	2	(R) − (Adr.) → R
Logisch verknüpfen	ANDZ	–	41	42	43				●	1	(RO) ∧ (R) → RO
	ANDI	44	45	46	47				●	2	(R) ∧ Wert → R
	ANDR	48	49	4A	4B				●	2	(R) ∧ (PC+V) → R
	ANDA	4C	4D	4E	4F				●	3	(R) ∧ (Adr.) → R
	IORZ	60	61	62	63				●	1	(RO) ∨ (R) → RO
	IORI	64	65	66	67				●	2	(R) ∨ Wert → R
	IORR	68	69	6A	6B				●	2	(R) ∨ (PC+V) → R
	IORA	6C	6D	6E	6F				●	3	(R) ∨ (Adr.) → R
	EORZ	20	21	22	23				●	1	(RO)−(R) → RO
	EORI	24	25	26	27				●	2	(R)−Wert → R
	EORR	28	29	2A	2B				●	2	(R)−(PC+V) → R
	EORA	2C	2D	2E	2F				●	3	(R)−(Adr.) → R
Rot	RRR	50	51	52	53	●	●	●	●	1	→(R)→
	RRL	D0	D1	D2	D3	●	●	●	●	1	←(R)←
Vergleichen	COMZ	E0	E1	E2	E3				●	1	(RO) ≷ (R)
	COMI	E4	E5	E6	E7				●	2	(R) ≷ Wert
	COMR	E8	E9	EA	EB				●	2	(R) ≷ (PC+V)
	COMA	EC	ED	EE	EF				●	3	(R) ≷ (Adr.)
	TMI	F4	F5	F6	F7				●	2	Alle«1»=1:00→CC

(R) = Inhalt eines Registers, V = Relative Verschiebung, STACK = Stapelspeicher, (Adr.) = Inhalt einer absoluten Adresse, PC = Programm-Adreßzähler, Port = Ein-/Ausgabe

Operationscode des 2650

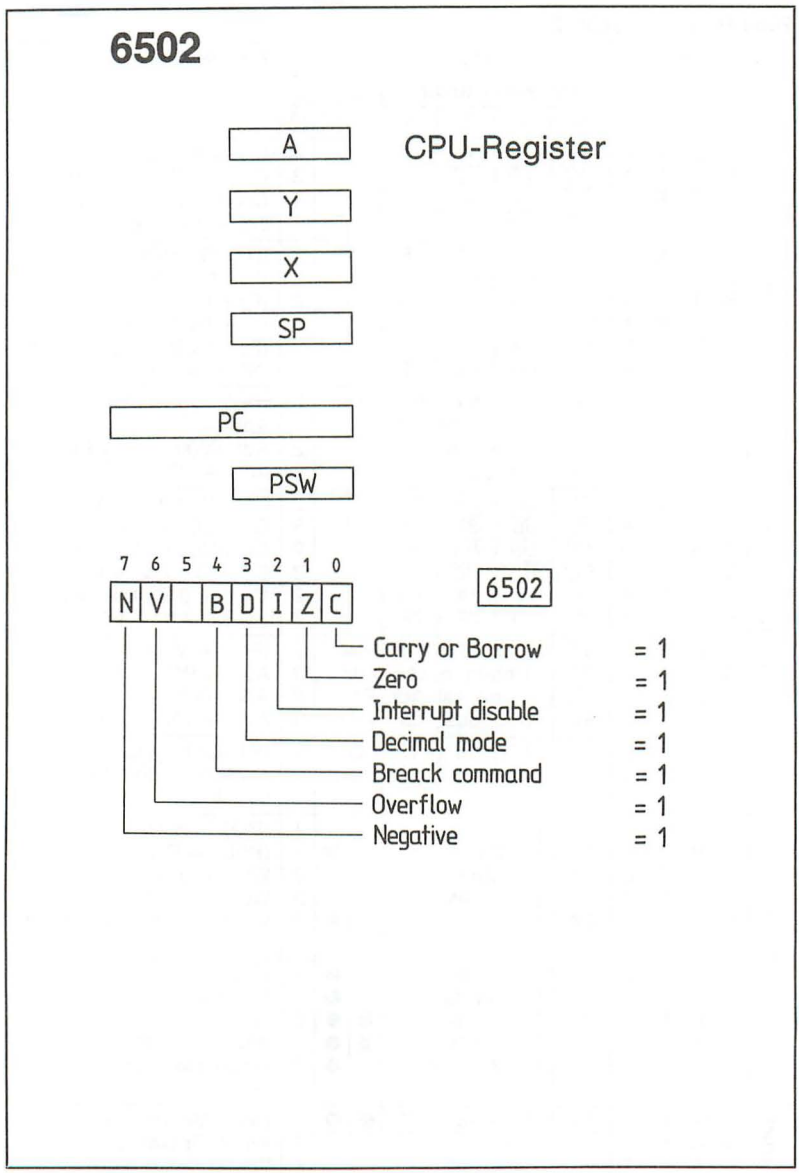
00	—	40	HALT	80	ADDZ, 0	C0	NOP
01	LODZ, 1	41	ANDZ, 1	81	ADDZ, 1	C1	STRZ, 1
02	LODZ, 2	42	ANDZ, 2	82	ADDZ, 2	C2	STRZ, 2
03	LODZ, 3	43	ANDZ, 3	83	ADDZ, 3	C3	STRZ, 3
04	LODI, 0	44	ANDI, 0	84	ADDI, 0	C4	—
05	LODI, 1	45	ANDI, 1	85	ADDI, 1	C5	—
06	LODI, 2	46	ANDI, 2	86	ADDI, 2	C6	—
07	LODI, 3	47	ANDI, 3	87	ADDI, 3	C7	—
08	LODR, 0	48	ANDR, 0	88	ADDR, 0	C8	STRR, 0
09	LODR, 1	49	ANDR, 1	89	ADDR, 1	C9	STRR, 1
0A	LODR, 2	4A	ANDR, 2	8A	ADDR, 2	CA	STRR, 2
0B	LODR, 3	4B	ANDR, 3	8B	ADDR, 3	CB	STRR, 3
0C	LODA, 0	4C	ANDA, 0	8C	ADDA, 0	CC	STRA, 0
0D	LODA, 1	4D	ANDA, 1	8D	ADDA, 1	CD	STRA, 1
0E	LODA, 2	4E	ANDA, 2	8E	ADDA, 2	CE	STRA, 2
0F	LODA, 3	4F	ANDA, 3	8F	ADDA, 3	CF	STRA, 3
10	—	50	RRR, 0	90	—	D0	RRL, 0
11	—	51	RRR, 1	91	—	D1	RRL, 1
12	SPSU	52	RRR, 2	92	LPSU	D2	RRL, 2
13	SPSL	53	RRR, 3	93	LPSL	D3	RRL, 3
14	RETC, *0*	54	REDE, 0	94	DAR, 0	D4	WRTE, 0
15	RETC, *1*	55	REDE, 1	95	DAR, 1	D5	WRTE, 1
16	RETC, *2*	56	REDE, 2	96	DAR, 2	D6	WRTE, 2
17	RETC, *3*	57	REDE, 3	97	DAR, 3	D7	WRTE, 3
18	BCTR, *0*	58	BRNR, 0	98	BCFR, *0*	D8	BIRR, 0
19	BCTR, *1*	59	BRNR, 1	99	BCFR, *1*	D9	BIRR, 1
1A	BCTR, *2*	5A	BRNR, 2	9A	BCFR, *2*	DA	BIRR, 2
1B	BCTR, *3*	5B	BRNR, 3	9B	ZBRR	DB	BIRR, 3
1C	BCTA, *0*	5C	BRNA, 0	9C	BCFA, *0*	DC	BIRA, 0
1D	BCTA, *1*	5D	BRNA, 1	9D	BCFA, *1*	DD	BIRA, 1
1E	BCTA, *2*	5E	BRNA, 2	9E	BCFA, *2*	DE	BIRA, 2
1F	BCTA, *3*	5F	BRNA, 3	9F	BXA	DF	BIRA, 3
20	EORZ, 0	60	IORZ, 0	A0	SUBZ, 0	E0	COMZ, 0
21	EORZ, 1	61	IORZ, 1	A1	SUBZ, 1	E1	COMZ, 1
22	EORZ, 2	62	IORZ, 2	A2	SUBZ, 2	E2	COMZ, 2
23	EORZ, 3	63	IORZ, 3	A3	SUBZ, 3	E3	COMZ, 3
24	EORI, 0	64	IORI, 0	A4	SUBI, 0	E4	COMI, 0
25	EORI, 1	65	IORI, 1	A5	SUBI, 1	E5	COMI, 1
26	EORI, 2	66	IORI, 2	A6	SUBI, 2	E6	COMI, 2
27	EORI, 3	67	IORI, 3	A7	SUBI, 3	E7	COMI, 3
28	EORR, 0	68	IORR, 0	A8	SUBR, 0	E8	COMR, 0
29	EORR, 1	69	IORR, 1	A9	SUBR, 1	E9	COMR, 1
2A	EORR, 2	6A	IORR, 2	AA	SUBR, 2	EA	COMR, 2
2B	EORR, 3	6B	IORR, 3	AB	SUBR, 3	EB	COMR, 3
2C	EORA, 0	6C	IORA, 0	AC	SUBA, 0	EC	COMA, 0
2D	EORA, 1	6D	IORA, 1	AD	SUBA, 1	ED	COMA, 1
2E	EORA, 2	6E	IORA, 2	AE	SUBA, 2	EE	COMA, 2
2F	EORA, 3	6F	IORA, 3	AF	SUBA, 3	EF	COMA, 3
30	REDC, 0	70	REDD, 0	B0	WRTC, 0	F0	WRTD, 0
31	REDC, 1	71	REDD, 1	B1	WRTC, 1	F1	WRTD, 1
32	REDC, 2	72	REDD, 2	B2	WRTC, 2	F2	WRTD, 2
33	REDC, 3	73	REDD, 3	B3	WRTC, 3	F3	WRTD, 3
34	RETE, *0*	74	CPSU	B4	TPSU	F4	TMI, 0
35	RETE, *1*	75	CPSL	B5	TPSL	F5	TMI, 1
36	RETE, *2*	76	PPSU	B6	—	F6	TMI, 2
37	RETE, *3*	77	PPSL	B7	—	F7	TMI, 3
38	BSTR, *0*	78	BSNR, 0	B8	BSFR, *0*	F8	BDRR, 0
39	BSTR, *1*	79	BSNR, 1	B9	BSFR, *1*	F9	BDRR, 1
3A	BSTR, *2*	7A	BSNR, 2	BA	BSFR, *2*	FA	BDRR, 2
3B	BSTR, *3*	7B	BSNR, 3	BB	ZBSR	FB	BDRR, 3
3C	BSTA, *0*	7C	BSNA, 0	BC	BSFA, *0*	FC	BDRA, 0
3D	BSTA, *1*	7D	BSNA, 1	BC	BSFA, *1*	FD	BDRA, 1
3E	BSTA, *2*	7E	BSNA, 2	BE	BSFA, *2*	FE	BDRA, 2
3F	BSTA, *3*	7F	BSNA, 3	BF	BSXA	FF	BDRA, 3

Kursiv geschriebene Ziffern hinter dem Kürzel bedeuten, daß der Bedingungscode die Ausführung des Befehls bestimmt; sonst bezeichnet die Ziffer das beteiligte Register

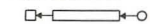
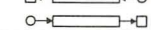
2650 Befehlstabelle 2

	Assembler	Register *Bedingungscode* 0	1	2	3	C	CC	Bytes	Befehlsablauf
Bedingtes Springen	BCTR	*18*	*19*	*1A*	*1B*			2	CC = CC: (PC)+V→PC
	BCTA	*1C*	*1D*	*1E*	*1F*			3	CC = CC: Adr. → PC
	BCFR	*98*	*99*	*9A*	—			2	CC ≠ CC: (PC)+V→PC
	BCFA	*9C*	*9D*	*9E*	—			3	CC ≠ CC: Adr. → PC
	BRNR	58	59	5A	5B			2	(R)≠0: (PC)+V→PC
	BRNA	5C	5D	5E	5F			3	(R)≠0: Adr.→PC
	BIRR	D8	D9	DA	DB			2	(R)+1≠0: (PC)+V→PC
	BIRA	DC	DD	DE	DF			3	(R)+1≠0: Adr.→PC
	BDRR	F8	F9	FA	FB			2	(R)−1≠0: (PC)+V→PC
	BDRA	FC	FD	FE	FF			3	(R)−1≠0: Adr.→PC
Unbed. Spr.	BCTR	1B	nach relativ. Adr.					2	(PC) + V → PC
	BCTA	1F	nach absolut. Adr.					3	Adr. → PC
	ZBRR	9B	n. rel. Adr. Page 1					2	Adr. 0000 + V → PC
	BXA	9F	n. abs. Adr. + IX					3	Adr. + (R3) → PC
Unterprogramme	BSTR	*38*	*39*	*3A*	*3B*			2	CC=CC: (PC)+V→PC
	BSTA	*3C*	*3D*	*3E*	*3F*			3	CC=CC: Adr.→PC
	BSFR	*B8*	*B9*	*BA*	—			2	CC≠CC: (PC)+V→PC
	BSFA	*BC*	*BB*	*BE*				3	CC≠CC: Adr.→PC
	BSNR	78	79	7A	7B			2	(R)≠0: (PC)+V→PC
	BSNA	7C	7D	7E	7F			3	(R)≠0: Adr.→PC
	BSTR	3B	Unbed. n. rel. Adr.					2	(PC) + V → PC
	BSTA	3F	Unbed. n. abs. Adr.					3	Adr. → PC
	ZBSR	BB	Unb. rel. Adr. P1					2	Adr. 0000 + V → PC
	BSXA	BF	Unb. abs. Adr. + IX					3	Adr. + (R3) → PC
	RETC	*14*	*15*	*16*	*17*	CC =		1	(STACK) → PC; SP −1
	RETE	*34*	*35*	*36*	*37*	CC		1	(STACK) → PC; SP −1; 0 → II
Programm-Status-Wort	LPSU	92	Alle Bit des oberen PSW CC = 00					1	(RO) → PSU
	SPSU	12					●	1	(PSU) → RO
	CPSU	74						2	Wo «I»: 0 → PSU
	PPSU	76						2	Wo «I»: I → PSU
	TPSU	B4					●	2	Wenn alle maskierten Bit = «1»
	LPSL	93	Alle Bit des unteren PSW CC = 00			●	●	1	(RO) → PSL
	SPSL	13					●	1	(PSL) → RO
	CPSL	75				●	●	2	Wo «I»: 0 → PSL
	PPSL	77				●	●	2	Wo «I»: I → PSL
	TPSL	B5					●	2	Wenn alle maskierten Bit = 1
Versch.	DAR	94	95	96	97	●	●	1	Dezimalkorrektur
	NOP	CO						1	Keine Operation
	HALT	40						1	Warten bis Reset

(R) = Inhalt eines Registers, V = Relative Verschiebung, STACK = Stapelspeicher, (Adr.) = Inhalt einer absoluten Adresse, PC = Programm-Adreßzähler, Port = Ein-/Ausgabe


6502
A
CPU-Register
Y
X
SP
PC
PSW
7 6 5 4 3 2 1 0
N V B D I Z C
6502
Carry or Borrow = 1
Zero = 1
Interrupt disable = 1
Decimal mode = 1
Breack command = 1
Overflow = 1
Negative = 1

6502 Befehlstabelle 1

Byte →	Unmittelbar 2	Absolut 3	Seite 0 2	(IND,X) 2	(IND),Y 2	Seite 0,X 2	Abs,X 3	Abs.Y 3	Seite 0,Y 2	N	V	Z	C	
L D A	A9	AD	A5	A1	B1	B5	BD	B9		●		●		A ← Oprd.
L D X	A2	AE	A6					BE	B6	●		●		X ← Oprd.
L D Y	A0	AC	A4			B4	BC			●		●		Y ← Oprd.
S T A		8D	85	81	91	95	9D	99						Oprd. ← (A)
S T X		8E	86						96					Oprd. ← (X)
S T Y		8C	84			94								Oprd. ← (Y)
A D C	69	6D	65	61	71	75	7D	79		●	●	●	●	(A) + Oprd. + C → A
S B C	E9	ED	E5	E1	F1	F5	FD	F9		●	●	●	●	(A) − Oprd. − C → A
A N D	29	2D	25	21	31	35	3D	39		●		●		(A) ∧ Oprd. → A
O R A	09	0D	05	01	11	15	1D	19		●		●		(A) ∨ Oprd. → A
E O R	49	4D	45	41	51	55	5D	59		●		●		(A) ∨ Oprd. → A
C M P	C9	CD	C5	C1	D1	D5	DD	D9		●		●	●	(A) − Oprd.
C P X	E0	EC	E4							●		●	●	(X) − Oprd.
C P Y	C0	CC	C4							●		●	●	(Y) − Oprd.
B I T		2C	24							B7	B6	●		(A) ∧ Oprd.
Byte →	Akku 1	Absolut 3	Seite 0 2			Seite 0, X 2	Abs. X 3	X 1	Y 1					
R O L	2A	2E	26			36	3E			●		●	●	
R O R	6A	6E	66			76	7E			●		●	●	
A S L	0A	0E	06			16	1E			●		●	●	
L S R	4A	4E	46			56	5E			0		●	●	
D E C		CE	C6			D6	DE			●		●		(Op.-Adr.) − 1 → Op.-Adr.
D E X								CA		●		●		(X) − 1 → X
D E Y									88	●		●		(Y) − 1 → Y
I N C		EE	E6			F6	FE			●		●		(Op.-Adr.) + 1 → Op.-Adr.
I N X								E8		●		●		(X) + 1 → X
I N Y									C8	●		●		(Y) + 1 → Y

Operationscode des 6502

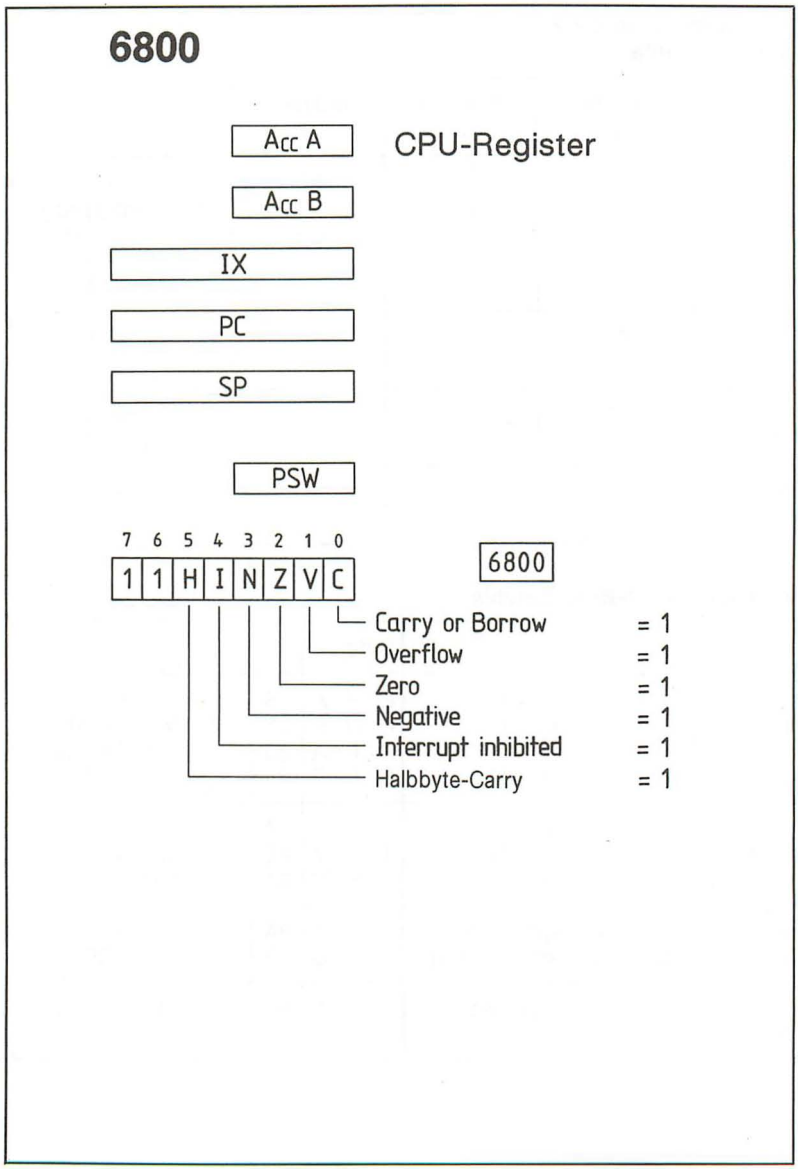
00 – BRK
01 – ORA – (Ind., X)
02 –
03 –
04 –
05 – ORA – S. Null
06 – ASL – S. Null
07 –
08 – PHP
09 – ORA – Imm.
0A – ASL – Accu
0B –
0C –
0D – ORA – Absolute
0E – ASL – Absolute
0F –
10 – BPL
11 – ORA – (Ind.), Y
12 –
13 –
14 –
15 – ORA – S. Null, X
16 – ASL – S. Null, X
17 –
18 – CLC
19 – ORA – Absolute, Y
1A –
1B –
1C –
1D – ORA – Absolute, X
1E – ASL – Absolute, X
1F –
20 – JSR
21 – AND – (Ind., X)
22 –
23 –
24 – BIT – S. Null
25 – AND – S. Null
26 – ROL – S. Null
27 –
28 – PLP
29 – AND – Imm.
2A – ROL – Accu
2B –
2C – BIT – Absolute
2D – AND – Absolute
2E – ROL – Absolute
2F –
30 – BMI
31 – AND – (Ind.), Y
32 –
33 –
34 –
35 – AND – S. Null, X
36 – ROL – S. Null, X
37 –
38 – SEC
39 – AND – Absolute, Y
3A –
3B –
3C –
3D – AND – Absolute, X
3E – ROL – Absolute, X
3F –
40 – RTI
41 – EOR – (Ind., X)
42 –
43 –
44 –
45 – EOR – S. Null
46 – LSR – S. Null
47 –
48 – PHA
49 – EOR – Imm.
4A – LSR – Accu
4B –
4C – JMP – Absolute
4D – EOR – Absolute
4E – LSR – Absolute
4F –
50 – BVC
51 – EOR – (Ind.), Y
52 –
53 –
54 –
55 – EOR – S. Null, X
56 – LSR – S. Null, X
57 –
58 – CLI
59 – EOR – Absolute, Y
5A –
5B –
5C –
5D – EOR – Absolute, X
5E – LSR – Absolute, X
5F –
60 – RTS
61 – ADC (Ind., X)
62 –
63 –
64 –
65 – ADC – S. Null
66 – ROR – S. Null
67 –
68 – PLA
69 – ADC – Imm.
6A – ROR – Accu
6B –
6C – JMP – Ind.
6D – ADC – Absolute
6E – ROR – Absolute
6F –
70 – BVS
71 – ADC – (Ind.), Y
72 –
73 –
74 –
75 – ADC – S. Null, X
76 – ROR – S. Null, X
77 –
78 – SEI
79 – ADC – Absolute, Y
7A –
7B –
7C –
7D – ADC – Absolute, X
7E – ROR – Absolute, X
7F –
80 –
81 – STA – (Ind., X)
82 –
83 –
84 – STY – S. Null
85 – STA – S. Null
86 – STX – S. Null
87 –
88 – DEY
89 –
8A – TXA
8B –
8C – STY – Absolute
8D – STA – Absolute
8E – STX – Absolute
8F –
90 – BCC
91 – STA – (Ind.), Y
92 –
93 –
94 – STY – S. Null, X
95 – STA – S. Null, X
96 – STX – S. Null, Y
97 –
98 – TYA
99 – STA – Absolute, Y
9A – TXS
9B –
9C –
9D – STA – Absolute, X
9E –
9F –
A0 – LDY – Imm.
A1 – LDA – (Ind., X)
A2 – LDX – Imm.
A3 –
A4 – LDY – S. Null
A5 – LDA – S. Null
A6 – LDX – S. Null
A7 –
A8 – TAY
A9 – LDA – Imm.
AA – TAX
AB –
AC – LDY – Absolute
AD – LDA – Absolute
AE – LDX – Absolute
AF –
B0 – BCS
B1 – LDA – (Ind.), Y
B2 –
B3 –
B4 – LDY – S. Null, X
B5 – LDA – S. Null, X
B6 – LDX – S. Null, Y
B7 –
B8 – CLV
B9 – LDA – Absolute, Y
BA – TSX
BB –
BC – LDY – Absolute, X
BD – LDA – Absolute, X
BE – LDX – Absolute, Y
BF –
C0 – CPY – Imm.
C1 – CMP) (Ind., X)
C2 –
C3 –
C4 – CPY – S. Null
C5 – CMP – S. Null
C6 – DEC – S. Null
C7 –
C8 – INY
C9 – CMP – Imm.
CA – DEX
CB –
CC – CPY – Absolute
CD – CMP – Absolute
CE – DEC – Absolute
CF –
D0 – BNE
D1 – CMP – (Ind.), Y
D2 –
D3 –
D4 –
D5 – CMP – S. Null, X
D6 – DEC – S. Null, X
D7 –
D8 – CLD
D9 – CMP – Absolute, Y
DA –
DB –
DC –
DD – CMP – Absolute, X
DE – DEC – Absolute, X
DF –
E0 – CPX – Imm.
E1 – SBC – (Ind., X)
E2 –
E3 –
E4 – CPX – S. Null
E5 – SBC – S. Null
E6 – INC – S. Null
E7 –
E8 – INX
E9 – SBC – Imm.
EA – NOP
EB –
EC – CPX – Absolute
ED – SBC – Absolute
EE – INC – Absolute
EF –
F0 – BEQ
F1 – SBC – (Ind.), Y
F2 –
F3 –
F4 –
F5 – SBC – S. Null, X
F6 – INC – S. Null, X
F7 –
F8 – SED
F9 – SBC – Absolute, Y
FA –
FB –
FC –
FD – SBC – Absolute, X
FE – INC – Absolute, X
FF –

6502 Befehlstabelle 2
Sprungbefehle

Byte →	Relativ 2	Absolut 3	Indirekt 3	
J M P J S R		4C 20	6C	Immer zum Unterprog.
B C C B C S	90 B0			C = 0 C = 1
B E Q B M I	F0 30			Z = 1 N = 1
B N E B P L	D0 10			Z = 0 N = 0
B V C B V S	50 70			V = 0 V = 1

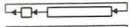
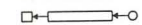
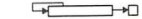
Verschiedene 1-Byte-Befehle

Byte →	1		Byte →	1	
C L C	18	0 → C	P H A	48	(A) → STACK
C L D	D8	0 → D	P H P	08	(PSW) → STACK
C L I	58	0 → I	P L A	68	A ← (STACK)
C L V	B8	0 → V	P L P	28	PSW ← (STACK)
S E C	38	1 → C	T A X	AA	(A) → X
S E D	F8	1 → D	T A Y	AB	(A) → Y
S E I	78	1 → I	T S X	BA	(SP) → X
R T I	40	Rückspr. v. Intrp.	T X A	8A	(X) → A
R T S	60	Rückspr. U.-Prog.	T X S	9A	(X) → SP
B R K N O P	00 EA	Haltpunkt PC + 1 → PC	T Y A	98	(Y) → A


6800
Acc A
CPU-Register
Acc B
IX
PC
SP
PSW
7 6 5 4 3 2 1 0
1 1 H I N Z V C
6800
Carry or Borrow = 1
Overflow = 1
Zero = 1
Negative = 1
Interrupt inhibited = 1
Halbbyte-Carry = 1

6800 Befehlstabelle 1

Byte →	Akkumulator A: Unmitt. 2	Seite 0 2	Index 2	Absolut 3	Akkumulator B: Unmitt. 2	Seite 0 2	Index 2	Absolut 3	H	N	Z	V	C	
LDA	86	96	A6	B6	C6	D6	E6	F6		●	●	0		A ← (Op.-Adr.)
STA		97	A7	B7		D7	E7	F7		●	●	0		(A) → Op.-Adr.
ADD	8B	9B	AB	BB	CB	DB	EB	FB	●	●	●	●	●	(A) + Operand → A
ADC	89	99	A9	B9	C9	D9	E0	F9	●	●	●	●	●	(A) + Oprd. + C → A
SUB	80	90	A0	B0	C0	D0	E0	F0		●	●	●	●	(A) − Operand → A
SBC	82	92	A2	B2	C2	D2	E2	F2		●	●	●	●	(A) − Oprd. − C → A
AND	84	94	A4	B4	C4	D4	E4	F4		●	●	0		(A) ∧ Oprd. → A
ORA	8A	9A	AA	BA	CA	DA	EA	FA		●	●	0		(A) ∨ Oprd. → A
EOR	88	98	A8	B8	C8	D8	E8	F8		●	●	0		(A) ∨ Oprd. → A
CMP	81	91	A1	B1	C1	D1	E1	F1		●	●	●	●	(A) − Oprd.
BIT	85	95	A5	B5	C5	D5	E5	F5						(A) ∧ Oprd.

Byte →	Akk. A 1	Akk. B 1	Index 2	Absolut 3	N	Z	V	C	
CLR	4F	5F	6F	7F	0	1	0	0	00 → (Op.-Adr.)
COM	43	53	63	73	●	●	0	1	$\overline{\text{Oprd.}}$ → Op.-Adr.
NEG	40	50	60	70	●	●			$\overline{\text{Oprd.}}$ + 1 → Op.-Adr.
DEC	4A	5A	6A	7A	●	●			Oprd. −1 → Op.-Adr.
INC	4C	5C	6C	7C	●	●			Oprd. +1 → Op.-Adr.
ROL	49	59	69	79	●	●		●	
ROR	46	56	66	77	●	●		●	
ASL	48	58	68	78	●	●		●	
ASR	47	57	67	77	●	●		●	
LSR	44	54	64	74	0	●		●	
TST	4D	5D	6D	7D	●	●	0	0	(Op.-Adr.) − 00

BYTE →	1	N	Z	V	C	
ABA	1B	Alle				(A) + (B) → A
SBA	10	●	●	●	●	(A) − (B) → A
DAA	19	●	●	●	●	(A) Dez.-Korr.
CBA	11	●	●	●	●	(A) − (B)
PSHA	36					(A) → ((SP))
PSHB	37					(B) → ((SP))
PULA	32					A → ((SP))
PULB	33					B ← ((SP))
TAB	16	●	●	0		(A) → B
TBA	17	●	●	0		(B) → A

Operationscode des 6800

00	*		40	NEG	A		80	SUB	A	IMM	C0	SUB	B	IMM
01	NOP		41	*			81	CMP	A	IMM	C1	CMP	B	IMM
02	*		42	*			82	SBC	A	IMM	C2	SBC	B	IMM
03	*		43	COM	A		83	*			C3	*		
04	*		44	LSR	A		84	AND	A	IMM	C4	AND	B	IMM
05	*		45	*			85	BIT	A	IMM	C5	BIT	B	IMM
06	TAP		46	ROR	A		86	LDA	A	IMM	C6	LDA	B	IMM
07	TPA		47	ASR	A		87	*			C7	*		
08	INX		48	ASL	A		88	EOR	A	IMM	C8	EOR	B	IMM
09	DEX		49	ROL	A		89	ADC	A	IMM	C9	ADC	B	IMM
0A	CLV		4A	DEO	A		8A	ORA	A	IMM	CA	ORA	B	IMM
0B	SEV		4B	*			8B	ADD	A	IMM	CB	ADD	B	IMM
0C	CLC		4C	INC	A		8C	CPX	A	IMM	CC	*		
0D	SEC		4D	TST	A		8D	BSR		REL	CD	*		
0E	CLI		4E	*			8E	LDS		IMM	CE	LDX		IMM
0F	SEI		4F	CLR	A		8F	*			CF	*		
10	SBA		50	NEG	B		90	SUB	A	DIR	D0	SUB	B	DIR
11	CBA		51	*			91	CMP	A	DIR	D1	CMP	B	DIR
12	*		52	*			92	SBC	A	DIR	D2	SBC	B	DIR
13	*		53	COM	B		93	*			D3	*		
14	*		54	LSR	B		94	AND	A	DIR	D4	AND	B	DIR
15	*		55	*			95	BIT	A	DIR	D5	BIT	B	DIR
16	TAB		56	ROR	B		96	LDA	A	DIR	D6	LDA	B	DIR
17	TBA		57	ASR	B		97	STA	A	DIR	D7	STA	B	DIR
18	*		58	ASL	B		98	EOR	A	DIR	D8	EOR	B	DIR
19	DAA		59	ROL	B		99	ADC	A	DIR	D9	ADC	B	DIR
1A	*		5A	DEC	B		9A	ORA	A	DIR	DA	ORA	B	DIR
1B	ABA		5B	*			9B	ADD	A	DIR	DB	ADD	B	DIR
1C	*		5C	INC	B		9C	CPX		DIR	DC	*		
1D	*		5D	TST	B		9D	*			DD	*		
1E	*		5E	*			9E	LDS		DIR	DE	LDX		DIR
1F	*		5F	CLR	B		9F	STS		DIR	DF	STX		DIR
20	BRA	REL	60	NEG		IND	A0	SUB	A	IND	E0	SUB	B	IND
21	*		61	*			A1	CMP	A	IND	E1	CMP	B	IND
22	BHI	REL	62	*			A2	SBC	A	IND	E2	SBC	B	IND
23	BLS	REL	63	COM		IND	A3	*			E3	*		
24	BCC	REL	64	LSR		IND	A4	AND	A	IND	E4	AND	B	IND
25	BCS	REL	65	*			A5	BIT	A	IND	E5	BIT	B	IND
26	BNE	REL	66	ROR		IND	A6	LDA	A	IND	E6	LDA	B	IND
27	BEQ	REL	67	ASR		IND	A7	STA	A	IND	E7	STA	B	IND
28	BVC	REL	68	ASL		IND	A8	EOR	A	IND	E8	EOR	B	IND
29	BVS	REL	69	ROL		IND	A9	ADC	A	IND	E9	ADC	B	IND
2A	BPL	REL	6A	DEC		IND	AA	ORA	A	IND	EA	ORA	B	IND
2B	BMI	REL	6B	*			AB	ADD	A	IND	EB	ADD	B	IND
2C	BGE	REL	6C	INC		IND	AC	CPX		IND	EC	*		
2D	BLT	REL	6D	TST		IND	AD	JSR		IND	ED	*		
2E	BGT	REL	6E	JMP		IND	AE	LDS		IND	EE	LDX		IND
2F	BLE	REL	6F	CLR		IND	AF	STS		IND	EF	STX		IND
30	TSX		70	NEG		EXT	B0	SUB	A	EXT	F0	SUB	B	EXT
31	INS		71	*			B1	CMP	A	EXT	F1	CMP	B	EXT
32	PUL	A	72	*			B2	SBC	A	EXT	F2	SBC	B	EXT
33	PUL	B	73	COM		EXT	B3	*			F3	*		
34	DES		74	LSR		EXT	B4	AND	A	EXT	F4	AND	B	EXT
35	TXS		75	*			B5	BIT	A	EXT	F5	BIT	B	EXT
36	PSH	A	76	ROR		EXT	B6	LDA	A	EXT	F6	LDA	B	EXT
37	PSH	B	77	ASR		EXT	B7	STA	A	EXT	F7	STA	B	EXT
38	*		78	ASL		EXT	B8	EOR	A	EXT	F8	ADC	B	EXT
39	RTS		79	ROL		EXT	B9	ADC	A	EXT	F9	ADC	B	EXT
3A	*		7A	DEC		EXT	BA	ORA	A	EXT	FA	ORA	B	EXT
3B	RTI		7B	*			BB	ADD	A	EXT	FB	ADD	B	EXT
3C	*		7C	INC		EXT	BC	CPX		EXT	FC	*		
3D	*		7D	TST		EXT	BD	JSR		EXT	FD	*		
3E	WAI		7E	JMP		EXT	BE	LDS		EXT	FE	LDX		EXT
3F	SWI		7F	CLR		EXT	BF	STS		EXT	FF	STX		EXT

A = Accumulator A, B = Accumulator B, REL = Relative, IND = Index, IMM = Immediate,
DIR = Direct = Seite Null, EXT = Extended = Absolut

Befehlstabelle 6800 2

Byte →	Unmittelb. 3	Seite 0 2	Index 2	Absolut 3	Register 1	N	Z	V	
CPX	8C	9C	AC	BC		●	●	●	(X) − Oprd.
DEX					09		●		(X − 1) → X
DES					34				(SP − 1) → SP
INX					08		●		(X + 1) → X
INS					31				(SP − 1) − SP
LDX	CE	DE	EE	FE		●	●	0	Oprd. → X
LDS	8E	9E	AE	BE		●	●	0	Oprd. → SP
STX		DF	EF	FF		●	●	0	(X) → Op.-Adr., Op.-Adr. + 1
STS		9F	AF	BF		●	●	0	(SP) → Op.-Adr., Op.-Adr. + 1
TXS					35				(X − 1) → SP
TSX					30				(SP + 1) → X
JMP			6E	7E					Adresse → PC
JSR			AD	BD					Adresse → PC; PC → ((Sp))

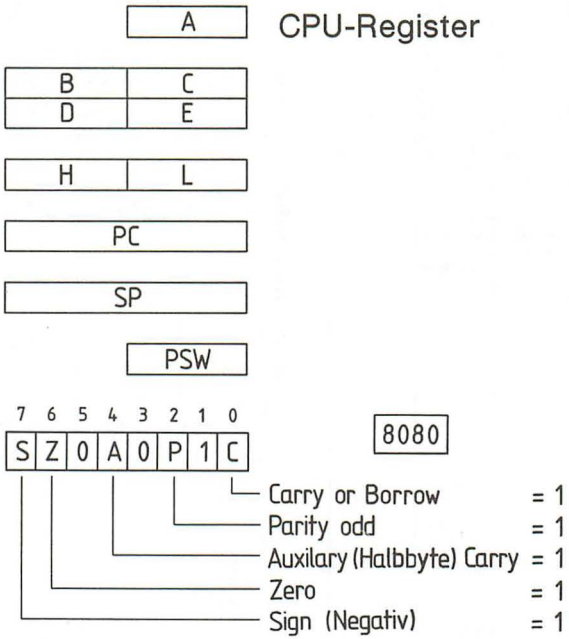
Relative Sprungbefehle

Byte →	Relativ 2	Sprung, wenn
BRA	20	immer
BCC	24	C = 0
BCS	25	C = 1
BEQ	27	Z = 1
BGE	2C	N oder V = 0
BGT	2E	Z + (N oder V) = 0
BHI	22	C + Z = 0
BLE	2F	Z + (N oder V) = 1
BLS	23	C + Z = 1
BLT	20	N oder V = 1
BMI	2B	N = 1
BNE	26	Z = 0
BVC	28	V = 0
BVS	29	V = 1
BPL	2B	N = 0
BSR	8D	(PC) + V → ((SP))

Verschiedene Ein-Byte-Befehle

Byte →	1	
CLC	0C	0 → C
CLI	0E	0 → I
CLV	0A	0 → V
SEC	0D	1 → C
SEI	0F	1 → I
SEV	0B	1 → V
TAP	06	(A) → PSW
TPA	07	(PSW) → A
NOP	01	(PC) + 1 → PC
RTI	3B	((SP)) → PC
RTS	39	((SP)) → PC
SWI	3F	Softw.-Interrupt
WAI	3E	Warten

8080


A
CPU-Register
B
C
D
E
H
L
PC
SP
PSW
7 6 5 4 3 2 1 0
S Z 0 A 0 P 1 C
8080
Carry or Borrow = 1
Parity odd = 1
Auxilary (Halbbyte) Carry = 1
Zero = 1
Sign (Negativ) = 1

8080 Registerbefehle

1-Byte-Befehle														
Reg. →	A	B	C	D	E	H	L	M	N	Z	H	P	C	
MOV A,R	7F	78	79	7A	7B	7C	7D	7E						A ← (R)
MOV B,R	47	40	41	42	43	44	45	46						B ← (R)
MOV C,R	4F	48	49	4A	4B	4C	4D	4E						C ← (R)
MOV D,R	57	50	51	52	53	54	55	56						D ← (R)
MOV E,R	5F	58	59	5A	5B	5C	5D	5E						E ← (R)
MOV H,R	67	60	61	62	63	64	65	66						H ← (R)
MOV L,R	6F	68	69	6A	6B	6C	6D	6E						L ← (R)
MOV M,R	77	70	71	72	73	74	75	76						M ← (R)
ADD R	87	80	81	82	83	84	85	86	●	●	●	●	●	(A) + (R) → A
ADC R	8F	88	89	8A	8B	8C	8D	8E	●	●	●	●	●	(A) + (R) + C → A
SUB R	97	90	91	92	93	94	95	96	●	●	●	●	●	(A) − (R) → A
SBC R	9F	98	99	9A	9B	9C	9D	9E	●	●	●	●	●	(A) − (R) − C → A
ANA R	A7	A0	A1	A2	A3	A4	A5	A6	●	●	●	●	●	(A) ∧ (R) → A
ORA R	B7	B0	B1	B2	B3	B4	B5	B6	●	●	●	●	●	(A) ∨ (R) → A
XRA R	AF	A8	A9	AA	AB	AC	AD	AE	●	●	●	●	●	(A) ⊻ (R) → A
CMP R	BF	B8	B9	BA	BB	BC	BD	BE	●	●	●	●	●	(A) − (R)
INR R	3C	04	0C	14	1C	24	2C	34	●	●		●	●	(R) + 1 → R
DCR R	3D	05	0D	15	1D	25	2D	35	●	●		●		(R) − 1 → R
	2-Byte-Befehl													
MVI R	3E	06	0E	16	1E	26	2E	36						R ← Wert

2-Byte-Befehle mit unmittelbarer Adressierung

		N	Z	H	P	C	
ADI	C6	●	●	●	●	●	(A) + Wert → A
ACI	CE	●	●	●	●	●	(A) + Wert + C → A
SUI	D6	●	●	●	●	●	(A) − Wert → A
SBI	DE	●	●	●	●	●	(A) − Wert − C → A
ANI	E6	●	●	●	●	●	(A) ∧ Wert → A
ORI	F6	●	●	●	●	●	(A) ∨ Wert → A
XRI	EE	●	●	●	●	●	(A) ⊻ Wert → A
CPI	FE	●	●	●	●	●	(A) − Wert

Restart-Befehle

		Adresse
RST 0	C7	0000
RST 1	CF	0008
RST 2	D7	0010
RST 3	DF	0018
RST 4	E7	0020
RST 5	EF	0028
RST 6	F7	0030
RST 7	FF	0038

Registerpaar-Befehle

Reg.-Paar Reg.	B BC	BYTE	D DE	BYTE	H HL	BYTE	SP	BYTE	PSW PSW. A	BYTE	C	
LXI	01	3	11	3	21	3	31	3				RP ← Wort
IXX	03	1	13	1	23	1	33	1				(RP) + 1 → RP
DCX	0B	1	1B	1	2B	1	38	1				(RP) − 1 → RP
DAD	09	1	19	1	29	1	39	1			●	(RP) + (HL) → HL
PUSH	C5	1	D5	1	E5	1			F5	1		(RP) → STACK
POP	C1	1	D1	1	E1	1			F1	1		RP ← (STACK)

Operationscode des 8080

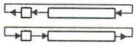
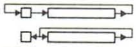
00	NOP		2B	DCX	H	56	MOV	D,M	81	ADD	C	AC	XRA	H	D7	RST	2
01	LXI	B,(16)	2C	INR	L	57	MOV	D,A	82	ADD	D	AD	XRA	L	D8	RC	
02	STAX	B	2D	DCR	L	58	MOV	E,B	83	ADD	E	AE	XRA	M	D9	...	
03	INX	B	2E	MVI	L,(8)	59	MOV	E,C	84	ADD	H	AF	XRA	A	DA	JC	Adr.
04	INR	B	2F	CMA		5A	MOV	E,D	85	ADD	L	B0	ORA	B	DB	IN	(8)
05	DCR	B	30			5B	MOV	E,E	86	ADD	M	B1	ORA	C	DC	CC	Adr.
06	MVI	B,(8)	31	LXI	SP,(16)	5C	MOV	E,H	87	ADD	A	B2	ORA	D	DD	...	
07	RLC		32	STA	Adr.	5D	MOV	E,L	88	ADC	B	B3	ORA	E	DE	SBI	D8
08			33	INX	SP	5E	MOV	E,M	89	ADC	C	B4	ORA	H	DF	RST	3
09	DAD	B	34	INR	M	5F	MOV	E,A	8A	ADC	D	B5	ORA	L	E0	RPO	
0A	LDAX	B	35	DCR	M	60	MOV	H,B	8B	ADC	E	B6	ORA	M	E1	POP	H
0B	DCX	B	36	MVI	M,(8)	61	MOV	H,C	8C	ADC	H	B7	ORA	A	E2	JPO	Adr.
0C	INR	C	37	STC		62	MOV	H,D	8D	ADC	L	B8	CMP	B	E3	XTHL	
0D	DCR	C	38			63	MOV	H,E	8E	ADC	M	B9	CMP	C	E4	CPO	Adr.
0E	MVI	C,(8)	39	DAD	SP	64	MOV	H,H	8F	ADC	A	BA	CMP	D	E5	PUSH	H
0F	RRC		3A	LDA	Adr.	65	MOV	H,L	90	SUB	B	BB	CMP	E	E6	ANI	(8)
10			3B	DCX	SP	66	MOV	H,M	91	SUB	C	BC	CMP	H	E7	RST	4
11	LXI	D,(16)	3C	INR	A	67	MOV	H,A	92	SUB	D	BD	CMP	L	E8	RPE	
12	STAX	D	3D	DCR	A	68	MOV	L,B	93	SUB	E	BE	CMP	M	E9	PCHL	
13	INX	D	3E	MVI	A,(8)	69	MOV	L,C	94	SUB	H	BF	CMP	A	EA	JPE	Adr.
14	INR	D	3F	CMC		6A	MOV	L,D	95	SUB	L	C0	RNZ		EB	XCHG	
15	DCR	D	40	MOV	B,B	6B	MOV	L,E	96	SUB	M	C1	POP	B	EC	CPE	Adr.
16	MVI	D,(8)	41	MOV	B,C	6C	MOV	L,H	97	SUB	A	C2	JNZ	Adr.	ED	...	
17	RAL		42	MOV	B,D	6D	MOV	L,L	98	SBB	B	C3	JMP	Adr.	EE	XRI	(8)
18			43	MOV	B,E	6E	MOV	L,M	99	SBB	C	C4	CNZ	Adr.	EF	RST	5
19	DAD	D	44	MOV	B,H	6F	MOV	L,A	9A	SBB	D	C5	PUSH	B	F0	RP	
1A	LDAX	D	45	MOV	B,L	70	MOV	M,B	9B	SBB	E	C6	ADI	(8)	F1	POP	PSW
1B	DCX	D	46	MOV	B,M	71	MOV	M,C	9C	SBB	H	C7	RST	0	F2	JF	Adr.
1C	INR	E	47	MOV	B,A	72	MOV	M,D	9D	SBB	L	C8	RZ		F3	DI	
1D	DCR	E	48	MOV	C,B	73	MOV	M,E	8E	SBB	M	C9	RET	Adr.	F4	CP	Adr.
1E	MVI	E,(8)	49	MOV	C,C	74	MOV	M,H	9F	SBB	A	CA	JZ		F5	PUSH	PSW
1F	RAR		4A	MOV	D,D	75	MOV	M,L	A0	ANA	B	CB			F6	ORI	(8)
20			4B	MOV	C,E	76	HLT		A1	ANA	C	CC	CZ	Adr.	F7	RST	6
21	LXI	H,(16)	4C	MOV	C,H	77	MOV	M,A	A2	ANA	D	CD	CALL	Adr.	F8	RM	
22	SHLD	Adr.	4D	MOV	C,L	78	MOV	A,B	A3	ANA	E	CE	ACI	(8)	F9	SPHL	
23	INX	H	4E	MOV	C,M	79	MOV	A,C	A4	ANA	H	CF	RST	1	FA	JM	Adr.
24	INR	H	4F	MOV	C,A	7A	MOV	A,D	A5	ANA	L	D0	RNC		FB	EI	
25	DCR	H	50	MOV	D,B	7B	MOV	A,E	A6	ANA	M	D1	POP	D	FC	CM	Adr.
26	MVI	H,(08)	51	MOV	D,C	7C	MOV	A,H	A7	ANA	A	D2	JNC	Adr.	FD		
27	DAA		52	MOV	D,D	7D	MOV	A,L	A8	XRA	B	D3	OUT	(8)	FE	CPI	D8
28			53	MOV	D,E	7E	MOV	A,M	A9	XRA	C	D4	CNC	Adr.	FF	RST	7
29	DAD	H	54	MOV	D,H	7F	MOV	A,A	AA	XRA	D	D5	PUSH	D			
2A	LHLD	Adr.	55	MOV	D,L	80	ADD	B	AB	XRA	E	D6	SUI	D8			

Adr. = 16-Bit-Adresse, (8) = 8-Bit-Datenbyte, (16) = 16-Bit-Datenwort

Sprungbefehle

Byte →	3		3		1	
Bedingung	Sprung		Unterprogramm		Rückkehr	
Immer	JMP	C3	CALL	CD	RET	C9
Z = 1	JZ	CA	CZ	CC	RZ	C8
Z = 0	JNZ	C2	CNZ	C4	RNZ	C0
C = 1	JC	DA	CC	DC	RC	D8
C = 0	JNC	D2	CNC	D4	RNC	D0
N = 1	JM	FA	CM	FC	RM	F8
N = 0	JP	F2	CP	F4	RP	F0
P = 1	JPE	EA	CPE	EC	RPE	E8
P = 0	JPO	E2	CPO	E4	RPO	E0

Akkumulator-Befehle

			N	Z	H	P	C	
LDA	3A	3						A ← (Adr.)
STA	32	3						Adr. ← (A)
LDAX B	0A	1						A ← ((BC))
LDAX D	1A	1						A ← ((DE))
STAX B	02	1						(BC) ← (A)
STAX D	12	1						(DE) ← (A)
IN	DB	2						A ← (Port)
OUT	D3	2						Port ← (A)
RAL	17	1					●	
RAR	1F	1					●	
RLC	07	1					●	
RRC	07	1					●	
CMA	2F	1						A ← ($\overline{A}$)
DAA	27	1	●	●	●	●	●	Dez.-Korr.
XRA A	AF	1	0	1	0	0	0	A ← 00
ORA A	B7	1	●	●	●	●	0	A ← (A)

Verschiedene 1-Byte-Befehle

STC	37	1 → C
CMC	3F	$\overline{C}$ → C
EI	FB	Intrpt. frei
DI	F3	Intrpt. gesp.
HLT	76	Halt
NOP	00	(PC) + 1 → PC

Registerpaar-HL-Befehle

LHLD	2A	3	HL ← (Adr.)
SHLD	22	3	Adr. ← (HL)
XCHG	EB	1	(HL) ↔ (DE)
PCHL	E9	1	(HL) ↔ (SP)
SPHL	F9	1	(HL) ↔ (PC)

Tabelle einiger derzeit am Markt befindlicher µP-Lern- und Übungsgeräte

Name	CPU	Tasten	Anzeige	Benötigte Spannung/Volt	Anwender-RAM	Betriebsprogramm	Ein-/Ausgabe	Gehäuse	Besonderheiten	Lieferant
Acorn	6502	16 Hex. 9 Funkt. einfach	8 15 Seg.	+8···12	1 K	Monitor (BASIC) (Assembler)	über Stecker 16	Nein	Zwei oder mehr Platinen, Europaformat, Alphanumerische Anzeige, Ergänzungsplatinen. Literatur in Englisch	⑮
ALPHA 1	6502	16 Hex. 11 Funkt., einfach	8 7 Seg.	Netzteil integriert	1,25 K	Monitor 2 K	über Stecker 16 4 LED	Ja	CPU-Karte Europaformat, viele Ausbaumöglichkeiten über Stecker anschließbar, 2 Rekorder anschließbar. Deutsche Literatur	⑨
AIM 65	6502	Schreibmaschinen-Tastatur	20 15 Seg.	+5 +24	1 K	Monitor Editor 4 K	über Stecker	Nein	Eingebauter alphanumerischer Drucker. Erweiterbar: BASIC-Interpreter, Assembler, 2 Rekorder anschließbar. Handbuch in Deutsch	⑭
Christiani	SC/ MP II	16 Hex. 17 Funkt., einfach	9 7 Seg.	Netzteil im Gehäuse	0,5 K	Monitor 1 K	über Erweiter.-Platinen	Ja	Gerät wird mit Fernstudienlehrgang geliefert. Anschluß von Erweiterungsplatinen (Drucker, PROM-Programmiergerät)	①

ECB 85	8085	16 Hex. 6 Funkt. z.T. doppelt	8 7 Seg.	+5	1,25 K	Monitor 2 K	Rasterfeld auf der Platine	Nein	Auch als Single-Board-Computer, Doppel-Europaformat, EPROM-Programmiermöglichkeit, Rasterfeld für Hardware-Erweiterungen	⑯
EKF-System	6800	16 Hex. 17 Funkt. 8 Anwend.	10 7 Seg.	+5	0,5 K	Monitor 2 K	Zusatzplatine	Nein	Systemplatinen, Europaformat in vielen Ausführungen, auch für praktische Anwendungen. Sehr komfortabler Monitor. Deutsche Anleitung	②
ESCO 1 und 2	6502	16 Hex. 7 Funkt., einfach	6 7 Seg.	+5 −5 (+12)	0,5 K	Monitor 2 K	über Stecker	Nein	Systemplatine im Europaformat und zusätzliche Monitoreinheit mit Anzeige und Tastatur	⑬
EURO-COM	6800	16 Hex. 14 Funkt., einfach	8 7 Seg.	+ 5 − 5 +12 −12	1 K	Monitor 2 K	Rasterfeld auf der Platine	Nein	40 Ein-/Ausgabeleitungen auf der Platine, Videokarte, ASCII-Tastatur als Zusatz, komfortabler Monitor. Deutsche Unterlagen	③
IN-STRUC-TOR 50	2650	16 Hex. 12 Funkt., einfach	8 7 Seg.	Netzteil integriert	0,5 K	Monitor 2 K	8 Schalter, 8 LED für 3 PORT	Ja	Direkte Schalter und Anzeigen für Ein-/Ausgabe und Interrupt-Darstellung. S-100-Bus-Stecker für Erweiterungen. Einfache Bedienung. Deutsche und englische Literatur	⑱

Name	CPU	Tasten	Anzeige	Benötigte Spannung/Volt	Anwender-RAM	Betriebsprogramm	Ein-/Ausgabe	Gehäuse	Besonderheiten	
ITT-Experimenter	8080	16 Hex. 21 Schalter	6 7 Seg. 16 LED	Netzteil integriert	0,5 K	Simulator, Monitor, 2 K	16 Schalt., 16 LED	Ja	Lehrgerät mit stufenweiser Einführung in die µP-Technik und sehr ausführlichen deutsch geschriebenen Lehrheften, ausbaufähig	⑦
HEATH-KIT	6800	17 Dopp.-Fkt.	6 7 Seg.	Integriertes Netzteil	0,25 K (0,50 K)	Monitor 1 K	über Stecker	Ja	Wird als Bausatz mit ausführlicher, deutsch übersetzter Bauanleitung geliefert	④
ICS 525 A	8080	16 Hex. 9 Funkt., z.T. doppelt	8 7 Seg.	mit Netzteil	2 K	Monitor 1 K (4 K)	über Stecker	Nein	Sehr umfangreiches deutsch übersetztes Lehrbuch für das Selbststudium, Platinen für Systemerweiterung lieferbar	⑤
KIM	6502	16 Hex., 7 Funkt., einfach	6 7 Seg.	+ 5 +12	1 K	Monitor 2 K	über Stecker	Nein	Sehr bekanntes Gerät, zu dem es viele Applikationsbeispiele gibt. Gehäuse mit Netzteil lieferbar. Englische Dokumentation	⑬
MEK 6800	6800	16 Hex. 8 Funkt., einfach	6 7 Seg.	+5	0,25 K (0,5 K)	Monitor 2 K	Rasterfeld auf einer Platine	Nein	Lieferbar als Bausatz für 2 Platinen mit deutsch übersetzter Beschreibung. Schaltplan und Monitor-Programmliste	⑩

MIKRO-SET	8080	16 Hex. 9 Funkt., einfach	6 7 Seg.	integriertes Netzteil	0,5 K	Monitor 1 K	über Stecker	Ja	Komplettgerät, deutsch übersetzte Beschreibung, weitere deutsche Literatur vom Hersteller erhältlich	⑯
NAS-COM 1	Z 80	Schreibmaschinen-Tastatur	Fernsehgerät	+ 5 − 5 +12 −12	1 K	Monitor Video 2 K	über Stecker	Nein	Anzeige über jedes Fernsehgerät, Eingabe über Volltastatur, verschiedene Erweiterungen, BASIC-Interpreter. Deutsche Beschreibung	⑪
NEC TK 80	8080	16 Hex. 9 Funkt., einfach	8 7 Seg.	+ 5 +12	0,5 K	0,75 K	Rasterfeld auf der Platine	Nein	Nur eine Platine, Speicherung des Anwenderprogramms in batteriegepuffertem C-MOS-RAM. Deutsch übersetztes Handbuch	⑫
SDK 85	8085	16 Hex. 8 Funkt., doppelt	6 7 Seg.	+5	0,5 K	2 K	Rasterfeld auf der Platine	Nein	Wird als Bausatz geliefert mit meist englischer Literatur. Großes Rasterfeld für Anwender-Hardware (Wire wrap)	⑥
TM 990/189	9980	45 Doppelfunktionen	10 7 Seg.	+ 5 +12 −12	1 K	Assembler 4 K	4 LED	Nein	Programmierung des 16-Bit-Prozessors in Assemblersprache, Lautsprecher auf der Platine	⑰
Z 80 KIT	Z 80	16 Hex. 12 Funkt., z.T. doppelt	8 7 Seg.		0,25 K	1 K	über Stecker	Nein	Verschiedene Erweiterungsplatinen, z.B. BASIC-Interpreter mit 4 K-RAM, Video, 20 Bit, parallele Ein-/Ausgabe	⑧

Einige Bezugsquellen für µP-Lern- und Übungsgeräte

① Dipl.-Ing. P. Christiani, 7750 Konstanz.
② EKF Elektronik – Meßtechnik OHG, Ehinger Straße 113, 4100 Duisburg 28.
③ ELTEC Elektronik GmbH, Neubrunner Straße 10, 6500 Mainz.
④ HEATHKIT Bausatz-Elektronik GmbH, Robert-Bosch-Straße 32, 6072 Dreieich-Sprendlingen.
⑤ ICSD Publishing GmbH, Leonardstraße 58, 8000 München 19.
⑥ Intel Semiconductor, Seidlstraße 27, 8000 München 2.
⑦ ITT-Fachlehrgänge, Postfach 15 70, 7500 Pforzheim.
⑧ Kontron Elektronik GmbH, Breslauer Straße 2, 8057 Eching.
⑨ MCS, Tennstedter Straße 18, 1000 Berlin 46.
⑩ Motorola, Münchner Straße 18, 8043 Unterföhrung.
⑪ NAS Elektronik GmbH, Brienner Straße 56, 8000 München 2.
⑫ NEC Electronics (Europa) GmbH, Immermannstraße 22, 4000 Düsseldorf 1.
⑬ Neumüller, Eschenstraße 2, 8021 Taufkirchen.
⑭ Rockwell, Fraunhoferstraße 119, 8033 Martinsfeld.
⑮ J. Schumpich, Postfach 17 20, 8021 Taufkirchen.
⑯ Siemens AG, Balanstraße 73, 8000 München 80.
⑰ Texas Instruments, Haggertystraße 1, 8050 Freising.
⑱ Valvo, Burchhardstraße 19, 2000 Hamburg 1.

Erklärung der einzelnen Befehle

2650
ADD 141
AND 141
BCT 166, 168
BCF 168
BDR 168
BIR 168
BSF 171
BSX 171
BST 171
BXA 166
COM 156
CPS 161
DAR 174
EOR 141
HALT 175
IOR 141
LOD 119, 122, 128, 131
LPS 161
NOP 175
PPS 161
RED 153
RET 172
ROL 150
ROR 150
STR 119, 128, 131
SPS 161
SUB 141
TMI 156
TPS 161
WRT 153
ZBR 166
ZBS 172

6502
ADC 142
AND 142
ASL 150
BCC 167
BEQ 167
BIT 158
BMI 167
BNE 167
BPL 167
BRK 175
BVC 167
BVS 167
CLC 160
CLD 160
CLI 160
CLV 160
CMP 155
CPX 155
CPY 155
DEC 147
DEX 147
DEY 147
EOR 142
INC 147
INX 147
INY 147
JMP 166
JSR 170
LDA 122, 125, 12
LDX 122, 125, 129
LDY 122, 125, 129
LSR 150
NOP 175
ORA 142
PHA 163
PHP 163
PLA 163
PLP 163
ROL 150
ROR 150
RTI 170
RTS 170
SBC 142
SEC 160
SED 160
SEI 160

STA 125
STX 125
STY 125
TAX 118
TAY 118
TSX 118
TXA 118
TYA 118

6800
ADD 143
ABA 143
ADC 143
AND 143
ASL 151
ASR 151
BIT 158
BRA 166
BCC 167
BCS 167
BEQ 167
BGE 167
BGT 167
BHI 167
BLE 167
BLS 167
BLT 167
BMI 167
BNE 167
BVC 167
BVS 167
BPL 167
BSR 170
CBA 155
CLC 160
CLI 160
CLR 148
CLV 160
CMP 155
COM 148
CPX 155
DAA 174
DEC 148
DES 148
DEX 148
EOR 143
INC 148
INS 148
INX 148
JMP 166
JSR 170
LDA 122, 125, 129
LDS 122, 126, 130
LDX 122, 126, 130
LSR 151
NEG 148
NOP 175
ORA 143
PSH 163
PUL 163
ROL 151
ROR 151
RTI 170
RTS 170
SBA 143
SBC 143
SEC 160
SEI 160
SEV 160
STA 125, 128
STS 130
STX 130
SUB 143
TAB 118
TAP 160
TBA 118
TPA 160
TST 158
TSX 343
TXS 343
WAI 175

8080
ACI 145
ADC 145
ADD 145
ADI 145
ANA 145
ANI 145
CALL 171
CC 171
CM 171
CMA 149
CMC 171
CMP 155
CNC 171
CNZ 171
CP 171
CPE 171
CPI 155
CPO 171
DAA 174
DAD 145
DCR 149
DCX 149
DI 175
EI 175
HLT 175
IN 153
INR 149
INX 149
JC 167
JM 167
JMP 166
JNC 167
JNZ 167
JP 167
JPE 167
JPO 167
LDA 126
LDAX 133
LXI 122
LHLD 126
MOV 120
MVI 122
NOP 175

ORA 145
ORI 145
OUT 153
POP 164
PUSH 164
RAL 152
RAR 152
RC 171
RET 171
RLC 152
RM 171
RNC 171
RNZ 171
RP 171
RPE 171
RPO 171
RRC 152
RST 177
SBC 145
SBI 145
SHLD 126
SPHL 121
SUB 145
SUI 145
STA 126
STAX 133
STC 160
XCHG 121
XRA 145
XRI 145
XTHL 121

Stichwortverzeichnis

A
Ablaufsteuerung 40
Absolute Adressierung 99, 123
ACIA 109
Addition 42, 87, 184, 247
Adresse 22, 34, 96
Adreßbank 263
Adreßbus 24
Adreßfeld 33
Adressierung 96
Adreßteile 100
Akkumulator 38
Algorithmus 291
AND 91
Anwenderprogramm 298
Arbeitsregister 38, 271
Arbeitsspeicher 280, 281
Architektur 271
Argumentenfeld 76
Arithmetik-Logik-Einheit 40
ASCII-Code 31, 217
Assemblieren 295
Assemblersprache 70, 96
Aufgabenbeschreibung 289
Ausblenden 91
Autoincrement 107

B
BASIC 72
Basis 65
Basisadresse 106
Baumstruktur 302
BCD-System 28, 60
Bedingung 75
Bedingungscode 141, 168
Baudrate 108
Befehl 73
Befehlsausführung 35
Befehlsadreßspeicher 38
Befehlsbus 24
Befehlsdecoder 40
Befehlsdiagramm 292
Befehlsformat 75
Befehlssatz 29, 73
Befehlssymbole 114
Befehlstabelle 73
Befehlscyclus 36
Betriebsprogramm 297
Bildschirmgerät 280
Binär-Dezimal-Wandlung 66
Binär-Multiplikation 57, 308
Binärsystem 24
BINARY CODED DECIMAL 28
BINARY DIGET 29

Binden 298
Bit 29
Bitmuster 31
Bitzählung 30
BLOCK 232
BLACK BOX 20
Borgen 44, 52
BORROW 44, 52
BRANCH 82, 165, 181, 277
BREAKPOINT 286, 300
BUS 24, 32
Byte 29

C
CALL 85
Carry 42, 85, 88
CHIP ENABLE 33
CHIP SELECT 33, 35
CLEAR 81, 85
CLOCK 23
COBOL 72
Codieren 295
Codewandlung 66, 217
COMPILER 72, 279
Computer-Platine
CONDITION CODE 115, 141, 144, 157
CONTROL-REGISTER 109
CRV 277
CRT 280
CPV 24
CYCLE 37

D
Datenblock 105, 225, 232
Datenbus 24
Datenfeld 105, 225, 232, 247
Datenrichtungsregister 109, 211, 215
Datensammelleitung 24
Datentransport 77, 118
Datenübernahme 111
Datenverarbeitung 79, 87, 139
DEBUGGING 281, 301
Decodierschaltung 32
Decrementieren 81, 146
DESTINATION 78
Dezimal-Binär-Wandlung 67
Dezimalkorrektur 61, 173, 252
Dezimalsystem 25, 42
DIREKT ADDRESSING 123, 276
DISABLE 110
Diskettenspeicher 280
Division 56
Divisionsmethode 67
Dokumentation 301
Doppelte Genauigkeit 54
DOUBLE PRECISION 54
DRAIN 78
Dualsystem 24

E
Editieren 295
EDITOR 297
Effektive Adresse 102
Ein-/Ausgabeeinheiten 21, 108
Ein-/Ausgabebefehle 107, 152
Ein-Chip-Computer 261
Einerkomplement 55
Einsprungadresse 239
ENABLE 72, 110
Entwicklungsgerät 279, 297
EQUATE 225
EPROM 21
EVEN PARITY 86
EXCHANGE 77
EXCLUSIVE OR 91
Exponentielle Zahlendarstellung 65
EXTENDED ADDRESSING 123

F
Fallabfrage 182
FALSE 84, 169
Fehlerbeseitigung 299
Fehlersuchprogramm 281, 301

Festwertspeicher 21
FILE NUMBER 287
FIXED POINT 65
FLAG-Register 39, 81
Fließkomma-Rechnung 65
FLOATING POINT 65
Flußdiagramm 291
Folge 179
FORTRAN 72

G
Gepackte Zahlen 60
Gleitkomma-Rechnung 65

H
Halbbyte-Übertrag 61, 85, 88
Halbcarry-Bit 85, 88
Haltbefehl 175
Haltpunkt 286, 300
HANDSHAKE 108
Hardware 19, 290
Hexadezimalsystem 25
Hilfsregister 39, 199
Höchstwertigste Stelle 30

I
IMPLIED ADDRESS 97
IMMIDIATE ADDRESSING 99
Incrementieren 81, 146
Index-Adressierung 105, 133
Indexregister 40, 105, 133
Indirekte Adressierung 104, 131, 225
Indirekt-Bit 131
Informationsmenge 30, 45
INHERENT ADDRESSING 97
Initialisieren 184
INPUT/OUTPUT UNIT 21, 108
INSTRUCTION 73
INTEGER 45
INTERDIGIT CARRY 85
INTERMEDIATE CARRY 85
INTERRUPT 110
Interrupt Vector 176
Invertieren 49

J
Ja/Nein-Entscheidung 84, 181
JUMP 82, 165

K
Kellerspeicher 94
Kennummer 287
K = Kilo 33
Kodieren 279
Kommentar 326
Kompatibilität 113
Komplementieren 49
Komplementrechnung 47
Konstante 121
Kontrollregister 109, 211
Kürzel 70

L
LABEL 71
Laden 78
LEAST SIGNIFICANT 44, 124
Lern- und Übungsgeräte 282, 284
Lesen 33, 79
LIFO 94
LINE BY LINE ASSEMBLER 285
Liste 225
LOAD 78
Logische Verknüpfungen 90
Löschen 85
LOOP 128

M
MACRO 293
Mantisse 65
Marke 71
Maschinencyclus 35
Maschinensprache 29, 70
Maske 81, 91, 121

Maskieren 91, 110
Mehrbyte-Verarbeitung 52, 247
MEMORY MAPPED I/O 152
Merker 184
Minuend 63
MNEMONICS 70
MODE 96
Modul 298, 302
Monitor 282, 285
MOST SIGNIFICANT 49, 124
MOVE 78
Multiplex 317
Multiplikation 56

N
Name 71
Negativ-Bit 142, 143
Negativwert 49
Negieren 49
NESTED SUBROUTINE 95
Nibble 29
Nibbleteilung 324
Niederstwertigste Stelle 30, 44
Null-Bit 86
Nullsetzen 209, 223
Nur-Oder-Verknüpfung 91, 209

O
Oder-Verknüpfung 91
ODD PARITY 86
OFFSET 102
Oktal-System 25
Operandenfeld 72
Operation 37, 73
Operationscode 74, 76
Operationsfeld 72
OR 91
OVERFLOW 52, 86

P
PACKED NUMBERS 60
PAGE 35, 128
PAGE ZERO 35, 102
Parameter 193
Parallelverarbeitung 108
Parität 86
Periphere Geräte 107
PIO, PIA 109
POINTER 98, 104, 271
PORT 108
POSTINDEXING 138
PREINDEXING 138
Programmablaufbefehle 82, 165
Programm-Analyse 306
PROGRAM COUNTER 38
PROGRAM LISTING 301
Programmiersprachen 72, 279
Programm-Module 298
Programmstart 176
Programm-Status-Register 39, 81, 160
Programm-Status-Wort 85, 117
Programm-Unterbrechung 110
PROM 21
Prüfbefehl 81, 154, 210
Prüfen 325

Q
Quellenprogramm 297
Quellregister 76
Quittungsverkehr 108

R
RAM 21
RANDOM ACCESS MEMORY 21
READ 33, 79
READ ONLY MEMORY 21
Redundanz 69
REENTRANT 192
Register 38, 116
Register-Adressierung 75, 97, 118
Register-Bank 162, 195, 262
Registerpaar 98, 251
Relative Adressierung 102, 130

Relative Verschiebung 197
RELOCTABLE 104
RESET 176
RESTART 177
Retten 85
RETURN 85
ROM 21
Rotieren 81, 92, 149
Rücksprung 84

S
SAVE 192
Schieben 81, 92, 149
Schleife 179, 223, 226
Schleifenzähler 179
Schnittstelle 108, 289
Schreiben 33, 79
Schreib-/Lesespeicher 21
Schrittbetrieb 286, 299
Schrittweise Verfeinerung 303
Seite 35, 128
Seite Null 34, 101, 129
Serielle Arbeitsweise 108
Sensoren 107
SEQUENCE 179
Setzen 85
SHIFT 81, 91
Sieben-Segment-Anzeige 312
SIGN BIT 49
Software 19
Sortieren 202
SOURCE 78
SOURCE PROGRAM 297
Speichern 22
Speicherkapazität 33
Sprungbefehle 82, 165
Sprungberechnung 197
Sprunggrenzen 219
STACK 79, 93, 163
STACK POINTER 93, 163
Standardroutinen 41
Stapelspeicher 79, 93, 163, 195
Stapelzeiger 93, 163
STATE 32
STATUS-Register-Befehle 159
Steuerleitungen 24
STORE 78
Struktogramm 303
Strukturelemente 179
Strukturierte Programmierung 302
SUBROUTINE 93
Subtraktion 42, 89, 184
Subtraktionsmethode 67
Subtrahend 63
Suchschleife 233
SWAP 265
Symbolische Adresse 71, 193, 276
Syntax 183
System-Software 281

T
Tabelle 319
Takt 35
Taktgeber 23, 36
Tauschbefehle 78
Testadapter 281
Testbefehle 81, 154, 210
Testen 299
Tetrade 61
TIMER 262
TOP DOWN PROGRAMMING 303
TRANSFER 78
Torschaltung 33
Transferbefehle 77
TRUE 84, 169
Typenfamilien 113

U
Überfließen 52, 86, 89
Überlauf-Bit 52, 86
Übertrag 44, 51, 85, 88
Und-Verknüpfung 91
Ungültige Tetrade 61
Unmittelbare Adressierung 76, 99, 121

Unterprogramm 93, 169
USART 109

V
VALUE 121
Verschachteln 95, 181, 191
Verschiebbar 104
Verschiebung 102, 295
Verdoppelung 58
Vergleichsbefehle 81, 154
Verknüpfung 87
Verzweigungsbefehle 82
Vorzeichen-Bit 49

W
Wartebefehl 175
Warteschleife 314
Wertigkeit 24
WORKSHOP 282
WORKSPACE REGISTER 271
Wort 29, 272
WRITE 33, 79

Z
Zähler 46, 191
Zahlensysteme 24
Zeiger 98, 104
Zeilen-Assembler 285
Zeitgeber 262
Zeitverzögerung 191, 198
Zentraleinheit 19
ZERO 86
Zielregister 76
Ziffernsymbole 25
Zweier-Komplement 45, 49, 87
Zufallszahl 313

VOGEL-BUCHVERLAG WÜRZBURG

Kompetent für Technik · Elektronik · Management · Wirtschaft

CHIP WISSEN

ist die Buchreihe, mit der Sie Ihr Mikrocomputerwissen systematisch vertiefen können. Sie bringt alles, worauf es ankommt.

Für Einsteiger, Fortgeschrittene und Profis. Mit praxisnahen Anwendungsbeispielen.

Sacht, Hans-Joachim

Von der passiven zur aktiven Computerei

332 Seiten, 106 Abbild.
ISBN 3-8023-0665-1

Hardware, Software. Mit der persönlichen Computerei beginnen, BASIC-Programme schreiben, worauf man beim Kauf achten muß.

Sacht, Hans-Joachim

BASIC-Versionen im Vergleich

180 Seiten, zahlr. Abb.
ISBN 3-8023-0752-6

Für Computerfreunde, die das Programmieren aus Freude am Lösen von Problemen betreiben. Programmierer können so Programme ins eigene System übernehmen.

Pol, Bernd

Wie man in BASIC programmiert

368 Seiten, 16 Abbild.
ISBN 3-8023-0637-6

An zwei bis ins Detail ausgearbeiteten Fallstudien werden die Grundlagen des Programmierens verdeutlicht und die wichtigsten BASIC-Bestandteile besprochen.

Baumann, Rüdeger

Computerspiele und Knobeleien programmiert in BASIC

304 S., zahlr. Abbild.
ISBN 3-8023-0703-8

Anleitung zum schöpferischen Umgang mit dem Computer: Aus der Spielidee entwickelt sich die Spielstrategie.

Sacht, Hans-Joachim

Vom Problem zum Programm

328 Seiten, 108 Abbild.
ISBN 3-8023-0715-1

Dieses Buch beginnt dort, wo die meisten BASIC-Lehrbücher aufhören. Es bringt keine abstrakten Beispiele, um die Wirkung von BASIC-Befehlen zu zeigen, sondern erklärt, wie vorzugehen ist, um ein Problem durch Programmierung zu lösen. Es erläutert, welche Vorüberlegungen notwendig sind und wie ein Programm Schritt für Schritt entsteht. Die 50 Beispiele sind nach Sachgebieten geordneten und ausführlich erklärt.

VOGEL-BUCHVERLAG WÜRZBURG

Kompetent für Technik · Elektronik · Management · Wirtschaft

Wernicke, Joachim

Computer für den Kleinbetrieb

148 Seiten, 12 Abb.
ISBN 3-8023-0711-9
Unverzichtbare Grundlagen, so kaufen Sie ihren Computer ohne Risiko, der Computereinstieg im Betrieb: Die Büroarbeiten, „Kleingedrucktes" zum Computereinstieg, Anhang.

Willis, Jerry/Pol, Bernd

Was der Mikrocomputer alles kann

366 Seiten, 100 Abbild.
ISBN 3-8023-0643-0
Diese Einführung für alle, insbesondere für Nichttechniker und Anfänger, bringt in leicht faßbarer Form alle Grundlagen der Computerei.

Tatzl, Gerfried

Praktische Problemanalyse

ca. 300 Seiten, zahlr. Abbild.
ISBN 3-8023-0745-3
Ohne den Leser in ein enges Denkschema zu pressen, wird bei Wahrung eines Mindestmaßes an Systematik die kreative Seite angesprochen. Beispiele u.a.m.

Schmidt, Klaus-Jürgen
Renner, Gerhard

Mikrocomputer-Betriebssysteme CP/M, CDOS, DOS

152 Seiten, 32 Abbild.
ISBN 3-8023-0655-4
Zur Erfassung der Leistungsfähigkeit von µP-Systemen werden Minibetriebssysteme dargestellt, dann Befehle u.v.a

Zaks, Rodnay

CP/M-Handbuch

310 S., zahlr. Abbild.
3-8023-0704-6
Die Anwendungen des Control Program for Microprocessors (CP/M) sind ausführlich, von Operationen am System bis hin zu Problemlösungen, beschrieben. Zahlreiche Fotos veranschaulichen die Hardware.

Baumann, Rüdeger

Programmieren mit PASCAL

272 Seiten, zahlr. Abb.
ISBN 3-8023-0667-8
Eine Einführung für Schüler und Hobbyprogrammierer. Die Einzelkomponenten von PASCAL werden mit den Aufgaben schrittweise erarbeitet und durch Übungen gefestigt.

Baumann, Rüdeger

Spiel, Idee und Strategie programmiert in Pascal

336 Seiten, zahlr.Abbild.
ISBN 3-8023-0732-1
Spielerisches Lernen, Programme in Pascal zu gestalten und anzuwenden. Die Strategie zu kniffligen Problemen.

Senftleben, Dietrich

Programmieren mit Logo

Einstieg — Praxis — Arbeitshilfen
352 Seiten, zahlreiche Abbildungen
3-8023-0744-5
Logo ist die Programmiersprache für PC. Ihre Stärke liegt im funktionsorientierten Konzept.

CHIP. Das Mikrocomputer-Fachmagazin. Für alle, die mehr wissen wollen und mehr wissen müssen, wenn es um Mikrocomputer geht. In Hobby und Beruf. Denn **CHIP** bringt umfassend und verständlich alles worauf es ankommt: Hintergrundberichte, Beispiele, umfassende Anleitungen, Software, Nachrichten, Trends und Meinungen. Ausgabe für Ausgabe eine geballte Ladung an Wissenswertem und Wichtigem. Damit Sie Bescheid wissen und mitreden können.

CHIP gibt es monatlich im Zeitschriften- und Buchhandel. Oder einfach bestellen beim **CHIP**-Leserservice, Vogel-Verlag Postfach 67 40, D-8700 Würzburg 1

ISBN 3-8023-0644-9

Dieses Buch soll Anfängern, die noch nichts oder nur wenig vom Programmieren wissen, die Einarbeitung in die Technik des Programmierens erleichtern. Es wird aber ebenso jedem, der sich mit Technik und Naturwissenschaft beschäftigt und damit zwangsläufig mit dem Mikroprozessor zu tun hat oder bekommt, nützlich sein. Techniker und Ingenieure, die ihre Ausbildung abgeschlossen haben, bevor es Mikroprozessoren gab, finden den Anschluß.

VOGEL-BUCHVERLAG
WÜRZBURG

Kompetent für Technik · Elektronik
Management · Wirtschaft